Université de Nantes
Faculté des Sciences et Techniques
Département d'Informatique

Travail d'Etude et de Recherche :

Algorithmes d'Inférence Grammaticale pour le Traitement des Langues

Véronique Moriceau - Guillaume Pasquier

encadrés par Alexandre Dikovsky et Christian Retoré

Juin 2002

Véronique Moriceau – Guillaume Pasquier
(encadrés par Alexandre Dikovsky et Christian Retoré)
Algorithmes d'Inférence Grammaticale pour le Traitement des Langues

Juin 2002 par Véronique Moriceau et Guillaume Pasquier

Algorithmes d'Inférence Grammaticale pour le Traitement des Langues

Véronique Moriceau – Guillaume Pasquier
(encadrés par Alexandre Dikovsky et Christian Retoré)

moriceauveronique@oreka.com
gpasquier@online.fr

SOMMAIRE

INTRODUCTION 1

1 L'APPRENTISSAGE 2

1.1 FONDEMENTS : L'ACQUISITION DE LA LANGUE MATERNELLE PAR L'ENFANT. 2
1.2 CONTEXTE : LE MODELE DE GOLD. 3
1.3 PRESENTATION DES GRAMMAIRES CATEGORIELLES 4
1.3 PERSPECTIVES ET LIMITES 5

2 LES GRAMMAIRES CATEGORIELLES 6

2.1 DEFINITIONS ET PROPRIETES 6
2.2 GRAMMAIRES CATEGORIELLES ET GRAMMAIRES HORS-CONTEXTE 7
2.3 APPRENTISSAGE DES GRAMMAIRES CATEGORIELLES 8
2.3.1 Définitions préalables 8
2.3.2 L'algorithme RG 10

3 LES GRAMMAIRES DE LAMBEK 14

3.1 DEFINITIONS ET PROPRIETES 14
3.2 APPRENTISSAGE DES GRAMMAIRES DE LAMBEK 15
3.2.1 Définitions préalables 15
3.2.2 L'algorithme RLG 17

4 LES GRAMMAIRES CATEGORIELLES AVEC ITERATIONS 22

4.1 LES GRAMMAIRES DE DEPENDANCE 22
4.2 LES GRAMMAIRES CATEGORIELLES AVEC ITERATIONS 24
4.3 APPRENTISSAGE DES GRAMMAIRES CATEGORIELLES AVEC ITERATIONS 25
4.3.1 Définitions préalables 25
4.3.2 L'algorithme RG* 27

5 DETAILS D'IMPLEMENTATION ET MANUEL D'UTILISATION 31

5.1 CONTENU DES FICHIERS 31
5.1.1 Langages et librairies utilisés 31
5.1.2 Organisation des fichiers 32
5.2 REPRESENTATION DES ARBRES 32
5.2.1 Structures des arbres en XML 32
5.2.2 Conversion des arbres XML en CaML 33
5.2.3 Structure des arbres en CaML 34
5.3 REPRESENTATION DU LEXIQUE 35
5.3.1 Représentation du lexique en CaML 35
5.3.2 Persistance du lexique 35
5.4 REPRESENTATION DES REGLES ET DES CATEGORIES 35
5.4.1 Convention de représentation des règles 35
5.4.2 Représentation des catégories en CaML 36
5.5 QUELQUES EXPLICATIONS SUR L'IMPLEMENTATION DES DIFFERENTS ALGORITHMES 36
5.5.1 Algorithmes d'apprentissage 36
5.5.2 Présentation de notre implémentation de la normalisation 37
5.5.3 Présentation de notre implémentation de l'unification 37
5.5.4 Complexité 38

5.5.5 Exemple d'exécution pas à pas 39
5.6 PRESENTATION DE L'INTERFACE 42
5.6.1 Présentation de l'interface actuelle 42
5.6.2 Propositions pour une interface plus adaptée 45
5.6.3 Conséquences sur la persistance du lexique 46
6 JEUX D'ESSAI 47
6.1 ALGORITHME RG 47
6.2 ALGORITHME RLG 48
6.3 ALGORITHME RG* 49
6.4 CONTRE-EXEMPLES 50
6.5 REMARQUES 51
CONCLUSION 52
Bibliographie 53
A Manuel d'utilisation 54
B Listing commenté 55

Introduction

L'inférence grammaticale est une question générale qui apparaît dès que l'on souhaite trouver une grammaire engendrant un ensemble d'exemples, afin de décrire de manière compacte et intelligible les données observées et d'en exhiber les régularités. Cette situation se rencontre notamment dans l'étude du génome, le diagnostic et le traitement automatique des langues. C'est cette dernière application qui nous intéresse.

Il existe deux grands types d'approches pour le traitement automatique du langage naturel : la première est fondée sur des techniques statistiques ou probabilistes appliquées à de grands corpus, tandis que la seconde propose un traitement symbolique de la langue pour en exhiber les mécanismes calculatoires.

La présente étude s'inscrit dans cette seconde approche et, plus précisément, dans le cadre des grammaires catégorielles. Celles-ci sont particulièrement intéressantes :

- en effet, leur formalisme simple et précis permet d'en faire un traitement informatique efficace,
- de plus, leur forte lexicalisation leur confèrent d'importantes propriétés d'apprentissage,
- le lien avec la sémantique de Montague facilite le calcul de représentations sémantiques
- enfin, le lien étroit des grammaires catégorielles avec la logique en font un objet d'étude intéressant pour leurs propriétés mathématiques.

Notre travail a d'abord consisté à comprendre les différentes notions et en particulier les grammaires utilisées puis à implanter les algorithmes d'apprentissage (RG, RG* et RLG).

Dans ce rapport, nous nous attacherons dans un premier temps à présenter le contexte de l'étude en abordant les questions de l'acquisition de la langue maternelle par l'enfant, du modèle de Gold ainsi que des grammaires catégorielles. Nous présenterons ensuite formellement les différentes grammaires utilisées : les grammaires AB, les grammaires rigides de Lambek et les grammaires catégorielles avec itérations. Enfin, nous aborderons le détail de l'implémentation de ces différents algorithmes.

PARTIE 1

L’apprentissage

1.1 Fondements : l’acquisition de la langue maternelle par l’enfant

Notion de grammaire universelle (d’après [1], dû à Chomsky)

La modélisation du processus par lequel un enfant apprend sa langue maternelle constitue un problème fondamental pour les sciences cognitives. En effet, ce processus met en jeu des mécanismes complexes, notamment pour la maîtrise des structures grammaticales.

Les psychologues cogniticiens s’accordent à dire que l’apprentissage de la grammaire se fait sur la base d’*exemples positifs*, c’est à dire uniquement par des phrases correctes du langage provenant de l’environnement culturel de l’enfant. Au contraire, il ne semble pas que les *exemples négatifs* – ceux que l’environnement de l’enfant lui signale comme étant faux – jouent un rôle significatif dans cet apprentissage.

En outre, le nombre d’exemples suffisant à l’enfant pour son apprentissage est très faible comparé à la complexité de la grammaire que l’enfant acquiert. C’est pour cela qu’on en vient à supposer l’existence d’une *grammaire universelle* innée, dont chaque langue particulière est une spécialisation acquise par instanciation.

Qu’est-ce que l’inférence grammaticale ? (d’après [1])

Le processus d’apprentissage de sa langue maternelle par un enfant peut être vu comme une instance du problème plus général de l’*inférence grammaticale*. On ne considèrera ici que l’inférence fondée sur des *exemples positifs*.

L’*inférence grammaticale* est le processus par lequel on acquiert la totalité de la structure grammaticale d’un langage formel sur la base d’exemples.

1.2 Contexte : le modèle de Gold

Intérêt (d'après [3])

L'objectif d'une théorie formelle de l'apprentissage, comme celle de Gold, est de fournir un modèle informatique pour représenter ce processus complexe qui consiste à extraire d'un ensemble assez petit de données brutes (les phrases correctes soumises à l'enfant) une connaissance fortement structurée (la grammaire de la langue apprise).

Schéma du processus d'inférence grammaticale selon Gold, 1967 (d'après [3] et [1])

L'inférence grammaticale est considérée comme un *processus infini*.

Nature des entrées : flux infini de phrases
On peut formuler deux hypothèses sur ce flux infini de phrases exposées à l'apprenant :
(i) seules des phrases correctes du langage apparaissent dans le flux, puisque l'on apprend à partir d'exemples positifs uniquement,
(ii) toutes les phrases possibles du langage doivent apparaître dans le flux (qui doit donc être une *énumération* des éléments du langage).

Algorithme :
Pour chaque nouvelle phrase présentée à l'apprenant
En considérant la grammaire obtenue à cette étape (hypothèse)
Formulation par l'apprenant d'une nouvelle hypothèse sur la nature de la grammaire censée engendrer le langage auquel appartiennent toutes les phrases qu'il a vu jusqu'ici

Sortie :
Comme l'apprenant est exposé à un nombre infini de phrases, il va faire l'hypothèse d'un nombre infini de grammaires, éventuellement différentes.

Succès :
Le processus d'apprentissage est réalisé avec succès lorsque la grammaire faite comme hypothèse :
- ne change plus à partir d'un certain point,
- équivaut à la grammaire cible qui engendre le langage.

Notion d'identification à la limite

Généralement, il est impossible de savoir si la grammaire est apprise correctement à une étape donnée : du fait de la nature infinie du processus, on ne peut pas savoir si la grammaire va changer à une étape ultérieure. Gold ([3]) nomme ce critère d'apprentissage avec succès l'*identification à la limit*e.

Apprenabilité d'une classe de grammaires (d'après [3] et [1])

Considérant le critère d'identification à la limite, une classe de grammaires est dite *apprenable* si pour toute grammaire de cette classe, et toute énumération des phrases de cette grammaire, il existe un algorithme d'apprentissage qui apprend avec succès cette grammaire.

Ce critère signifie qu'on ne cherche pas particulièrement à savoir *quand* l'apprentissage est réussi. L'objectif est plutôt de produire des algorithmes dont on est sûr

qu'ils apprennent correctement une grammaire (ou une classe de grammaires) si celle-ci existe, *s'ils sont appliqués au flux infini de phrase*s.

1.3 Présentation des grammaires catégorielles

Elles s'inscrivent dans l'approche calculatoire et symbolique de la linguistique informatique (qui s'oppose à l'étude de grands corpus par des techniques statistiques ou probabilistes). Les grammaires catégorielles offrent une perspective de liens entre logique, langues et langages ([9]).

Ce sont des grammaires fortement lexicalisées (d'après [9])

Dans ce type de grammaires, les règles sont uniformes, données une fois pour toutes et indépendantes de la langue : les règles constituent une sorte de grammaire universelle à la Chomsky, les variations d'une langue à l'autre sont prises en compte par la seule variation des types que le lexique associe à des entrées lexicales.

Exemple : on considère les lexiques d'une même phrase en français et en latin :

Pierre aime les roses	*Petrus rosas amat*
Pierre : NP *aime* : (NP \ S) / NP *les* : D / NP *roses* : NP	*Petrus* : NP *amat* : NP \ (NP \ S) *rosas* : NP

On constate que les règles (symbolisées par / et \) sont les mêmes dans les deux langues mais le lexique diffère.

La lexicalisation n'est pas qu'un plus esthétique : cette approche est d'un grand intérêt pour l'apprentissage, sujet d'importance primordiale en intelligence artificielle, dans les sciences cognitives et, sous une forme plus spécialisée, en linguistique. Les grammaires lexicalisées présentent le grand avantage de ne pas avoir de règles à apprendre, puisque celles-ci sont fixes ; il suffit alors de déterminer l'entrée lexicale associée à un nouveau mot ou au nouvel usage d'un mot déjà présent dans le lexique.

Grammaires rigides vs grammaires k-valuées.

Nous nous intéressons ici aux grammaires catégorielles rigides pour lesquelles chaque entrée lexicale possède un et un seul type. A titre documentaire, on peut évoquer les grammaires *k*-valuées qui constituent une extension des grammaires rigides : à chaque entrée lexicale, elles associent au plus *k* types (*k* étant un entier naturel fixé).

Par la suite, nous aborderons les trois types de grammaires catégorielles rigides suivants :
- les grammaires AB (RG),
- les grammaires rigides de Lambek (RLG),
- les grammaires catégorielles avec itération (RG*).

Le principe de l'unification

En analysant différentes phrases, on obtient plusieurs types pour un même mot. Or les grammaires catégorielles rigides imposent un et un seul type par entrée lexicale. Ainsi, le mécanisme d'unification, qui constitue l'élément central de l'apprentissage, identifie les différents types associés à un même mot puis substitue un type par un autre.

1.4 Perspectives et limites

Réserves quant à la validité de modèles tels que celui de Gold (d'après [3])

Les hypothèses faites sur les exemples positifs sont très vraisemblables d'un point de vue cognitif : la notion de « tête » (mot du syntagme dont dépendent les autres) ou de « fonction » dans un syntagme est vite perçue par l'enfant. Il faut cependant rester prudent quant à la validité de l'emploi de telles méthodes.

En effet, lorsqu'on formalise un processus cognitif aussi complexe que celui de l'apprentissage par l'enfant de sa langue maternelle, le modèle obtenu ne peut être qu'une simplification. Par exemple, on ne peut pas représenter les motivations psychologiques de l'enfant.

En outre, étant donnée l'évidente différence de nature et de capacités entre le cerveau humain et le modèle informatique, il n'est pas évident que le processus de traitement informatique le plus adapté à cette problématique soit le même que celui du cerveau humain.

Enfin, l'étude du processus d'apprentissage réel montre que celui-ci se fait par restrictions successives plutôt que par généralisation comme dans le modèle de Gold.

Potentiel applicatif des grammaires catégorielles pour l'apprentissage (d'après [9])

Dans le cadre des grammaires AB, W. Buszkowski et G. Penn ([2]) ont montré qu'un algorithme d'unification sur les types permettait de construire la grammaire la plus générale engendrant les énoncés proposés comme exemples positifs, et ce en temps $n.log(n)$.

M. Kanazawa ([5]) a étendu cet algorithme au cas où le nombre de types que le lexique associe à un mot est borné par un entier k et a prouvé sa convergence. Bien qu'utilisable en pratique, cet algorithme introduit un facteur exponentiel dans le pire des cas.

D'une façon générale, ces techniques d'apprentissage fondées sur l'unification sont en fait génériques et permettent de traiter de nombreuses questions d'apprentissage comme l'a montré J. Nicolas ([7]). Ainsi, les techniques d'apprentissage de langages peuvent s'appliquer à la construction de grammaires du génome : il n'y a pas encore d'utilisation des grammaires catégorielles, mais comme le génome contient des structures parenthésées, ce pourrait être un succès pour les grammaires catégorielles. On peut enfin envisager d'utiliser ces algorithmes pour le problème de l'extraction de données.

PARTIE 2

Les grammaires catégorielles

Nous nous intéressons ici aux *grammaires catégorielles classiques* (ou *grammaires AB*). Ce type de grammaires s'appuie sur des règles de calcul logique ce qui facilite le traitement informatique et l'étude de leurs propriétés mathématiques.

A la différence d'autres grammaires, comme les grammaires hors-contexte, constituées d'un ensemble de règles qui génèrent toutes les phrases d'un langage donné à partir d'un symbole initial, les grammaires catégorielles reconnaissent une chaîne de symboles bien formée si et seulement si les types affectés aux symboles se réduisent en un type nommé *Pr* désignant les chaînes bien formées.

2.1 Définitions et propriétés

Les grammaires catégorielles sont définies par un lexique qui associe un ensemble de *catégories* (types) à chaque symbole terminal (mot).

Pour les dérivations, on utilise les deux règles logiques suivantes : / (élimination droite) et \ (élimination gauche).

Définition 2.1.1 (Règles de dérivation).

$a \;\; B/A \;\; A \;\; b \;\; \rightarrow \;\; a \;\; B \;\; b$

$a \;\; A \;\; A\backslash B \;\; b \;\; \rightarrow \;\; a \;\; B \;\; b$

ce qui signifie que si une expression de type B/A (resp. A\B) est suivie (resp. précédée) d'une expression de type A, alors on obtient une expression de type B.

Les catégories des mots appartenant au langage seront donc constituées de types primitifs et des opérateurs / et \ .

Définition 2.1.2 (Grammaire catégorielle). [Bar-Hillel, Ajdukiewicz]
Une grammaire catégorielle est un quadruplet (S , Pr, F, s) tel que :

- *S est un ensemble fini de symboles terminaux (le vocabulaire)*
- *Pr est un ensemble fini (les types primitifs)*
- *F est une fonction de S vers les sous-ensembles finis de Tp, où Tp est le plus petit ensemble tel que :*
  1. *Pr Í Tp*
  2. *Si A, B Î Tp, alors (A / B), (A \ B) Î Tp*

  (F associe à chaque mot du vocabulaire une catégorie)
- *s Î Pr est la catégorie qui définit les phrases correctes du langage*

Définition 2.1.3 (Langage engendré).
Le langage engendré par une grammaire catégorielle (S , Pr, F, s) est l'ensemble :

$$\{ a_1,\ldots, a_n \in \Sigma^* \mid \forall\ 1 \le i \le n,\ \exists A_i : F(a_i) = A_i \text{ et } A_1,\ldots, A_n \to^* s \}$$

avec $\to^*$ *la fermeture transitive et réflexive de* $\to$.

Exemple 2.1.4.
La grammaire suivante définie par son lexique engendre le langage $\{ ({}^n)^n \mid n > 0 \}$:

(: s / B
) : B, s \ B

Ci-dessous la dérivation de ((( ))) :

s / B, s / B, s / B, B, s \ B, s \ B ®
s / B, s / B, s, s \ B, s \ B ®
s / B, s / B, B, s \ B ®
s / B, s, s \ B ®
s / B, B ®
s

2.2 Grammaires catégorielles et grammaires hors-contexte (d'après [9])

Définition 2.2.1 (Grammaire hors-contexte).
Une grammaire hors-contexte (CFG) est définie par :

- *un ensemble* ***NT*** *de symboles non-terminaux (****S*** *étant le symbole initial)*
- *un ensemble* ***T*** *de symboles terminaux*
- *un ensemble fini de règles de production de la forme X® W avec X Î NT et W Î (T È NT)**

Une séquence *V Î (T È NT)** *se dérive immédiatement en* une séquence *W Î (T È NT)** lorsqu'il existe *W¢, W¢¢, W¢¢¢ Î (T È NT)** *et X Î NT* tels que :

- V = W′ X W″
- X → W″ est une règle de production
- W = W′ W″ W‴

La relation $\rightarrow$ est définie sur les séquences de $(T \cup NT)^*$ comme la fermeture réflexive et transitive de "*se dérive immédiatement en*".

Le langage engendré par une CFG est le sous-ensemble de T^* qui contient les séquences qui peuvent être dérivées de S.

Définition 2.2.2 (Equivalence faible).
Deux grammaires qui engendrent le même langage sont faiblement équivalentes.

On peut démontrer que :

- *Toute grammaire hors-contexte (en forme normale de Greibach) est faiblement équivalente à une grammaire catégorielle.*
- *Toute grammaire catégorielle est faiblement équivalente à une grammaire hors-contexte (en forme normale de Chomsky).*

Les grammaires hors-contexte fournissent notamment un outil très employé en informatique : les automates. Ainsi, on peut imaginer de mettre ces outils au profit des automates de reconnaissance d'arbres dans le cadre des grammaires catégorielles.

2.3 Apprentissage des grammaires catégorielles

2.3.1 Définitions préalables

Nous ne réalisons ici que l'apprentissage de grammaires catégorielles rigides pour permettre l'unification.

Définition 2.3.1.1 (Grammaire catégorielle rigide).
Une grammaire catégorielle est rigide si à chaque mot de son vocabulaire n'est associée qu'une seule catégorie.

Comme expliqué précédemment, l'apprentissage se fait sur des exemples positifs donnés (phrases bien formées) dont on doit reconstituer les arbres de dérivation. Un arbre de dérivation pour une grammaire catégorielle (Σ , Pr, F, s) est défini de la manière suivante :

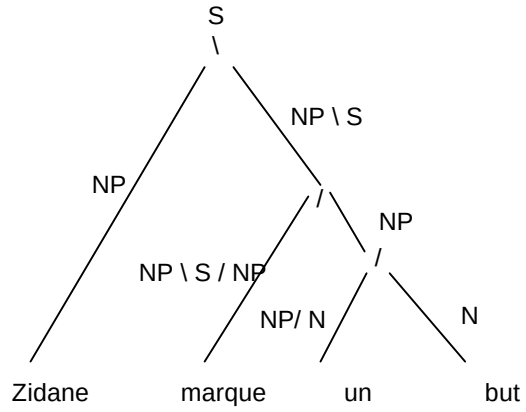
Définition 2.3.1.2 (Arbre de dérivation).
Si D est une dérivation de B à partir de $A_1,\ldots,A_n$ et $a_1,\ldots,a_n$ Î S tels que $G : a_i \mapsto A_i$ pour 1 £ i £ n, l'arbre constitué par D aux feuilles duquel sont attachés les $a_1,\ldots,a_n$ de gauche à droite dans cet ordre, est un arbre de dérivation partiel de G.

Un arbre de dérivation est complet si la catégorie qui figure à sa racine est s.

Les nœuds d'un arbre de dérivation sont marqués par un opérateur qui désigne la règle de réduction utilisée : / et \ dans les cas suivants :

- $B / A , A \Rightarrow_/ B$
- $A , A \setminus B \Rightarrow_\setminus B$

Exemple 2.3.1.3
Soit la phrase *Zidane marque un but*.
Son arbre de dérivation est :



Le but de l'apprentissage est de reconstituer, pour des exemples donnés, leur arbre de dérivation indiquant les catégories des mots. Pour cela, il faut que la structure des arbres soit connue : on suppose qu'elle sera fournie avec les exemples. Ceci constitue une approximation du modèle d'apprentissage.

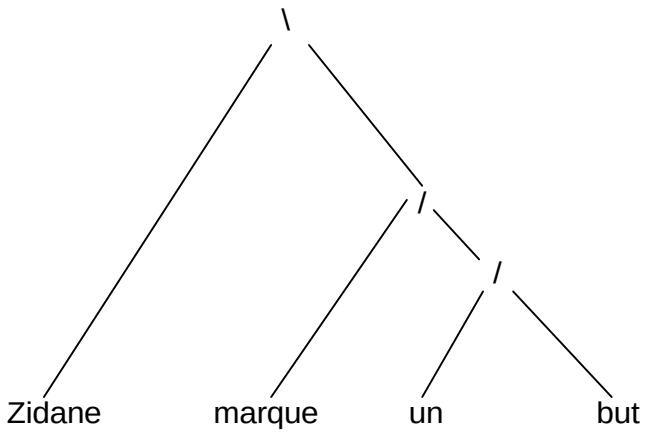
Définition 2.3.1.4 (Structure d'arbre de dérivation).
Une structure d'arbre de dérivation est un arbre de dérivation pour lequel les catégories des nœuds et des feuilles ne sont pas précisées.

La structure d'arbre de dérivation doit donc comprendre, pour chaque nœud, l'opérateur utilisé (/ ou \ ).

Exemple 2.3.1.5
Reprenons la phrase *Zidane marque un but*.
Sa structure d'arbre de dérivation peut être représentée de deux manières :

[\ Zidane [/ marque [/ un but]]]

ou



2.3.2 L'algorithme RG

L'algorithme utilisé pour réaliser cet apprentissage est l'**algorithme RG** de Buszkowski et Penn : il permet d'apprendre des grammaires catégorielles rigides à partir d'exemples positifs donnés sous forme de phrases avec leur structure d'arbre de dérivation.

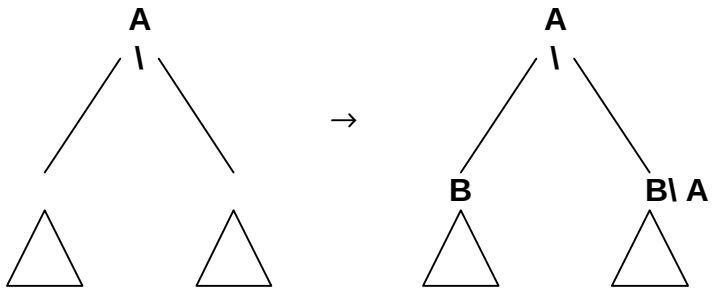
Il comprend deux phases :

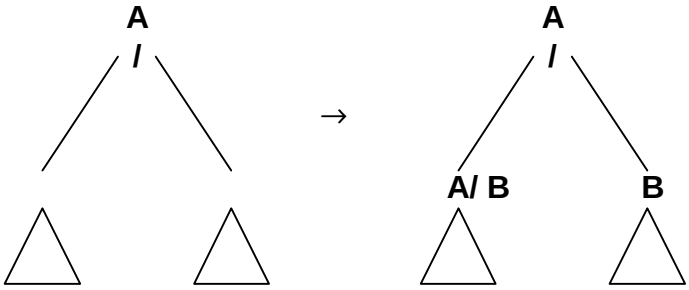
- la phase d'étiquetage
- la phase d'unification

Phase d'étiquetage :

La phase d'étiquetage consiste à déduire d'une structure d'arbre de dérivation un arbre de dérivation complet, c'est-à-dire à déduire les catégories des mots en connaissant les règles utilisées. Pour cela, on procède de la façon suivante :

1. Affecter la catégorie **s** à la racine de l'arbre puisque la phrase est correcte puis étiqueter les sous-arbres de la manière suivante :
2. Etiquetage d'un nœud de catégorie **A** :
   - si l'opérateur est **/** :
     1. affecter au nœud de l'opérande droit une nouvelle catégorie **B**
     2. affecter au nœud de l'opérande gauche la catégorie **A / B**
     3. étiqueter les sous-arbres
   - si l'opérateur est **** :
     1. affecter au nœud de l'opérande gauche une nouvelle catégorie **B**
     2. affecter au nœud de l'opérande droit la catégorie **B \ A**
     3. étiqueter les sous-arbres





Phase d'unification :

Si un mot est présent dans plusieurs exemples différents, la phase d'étiquetage précédente va lui affecter plusieurs catégories.

Comme la grammaire cible est rigide, il ne peut y avoir qu'une seule catégorie par mot. Si un mot a deux catégories distinctes, alors ces deux catégories sont obligatoirement identiques. Il faut donc identifier ces catégories et faire les substitutions nécessaires dans tout le vocabulaire : c'est l'unification.

Définition 2.3.2.1

Soient G_1, G_2 deux grammaires catégorielles, on dit que G_1 Í G_2 si et seulement si pour tout a Î S tel que $G_1 : a \mapsto A$, on a aussi $G_2 : a \mapsto A$.

Définition 2.3.2.2

Une substitution est une fonction s : Var ® Tp qui associe à toute variable un type.
On peut étendre cette fonction à s : Tp ® Tp avec, pour tous A, B Î Tp :

σ (A/ B) = σ (A) / σ (B)
σ (A\ B) = σ (A) \ σ (B)

Définition 2.3.2.3

Soit G = (S , Pr, F, s) une grammaire catégorielle, et s une substitution. Alors s[G] est la grammaire obtenue par application de s aux types de G :

σ [G] = (Σ , Pr, F, s)
σ [G] *est une instance par substitution de G.*

Définition 2.3.2.4

Soient deux types A et B. Une substitution s est un unifieur *de A et B si s (A) = s (B).*

Un unifieur s est un unifieur le plus général *de A et B si, pour tout autre unifieur t de A et B, il existe une substitution h telle que t = s* o*h , ie t (C) = h(s (C)) , pour C = A ou C = B.*

2.3.2.5 Algorithme d'unification

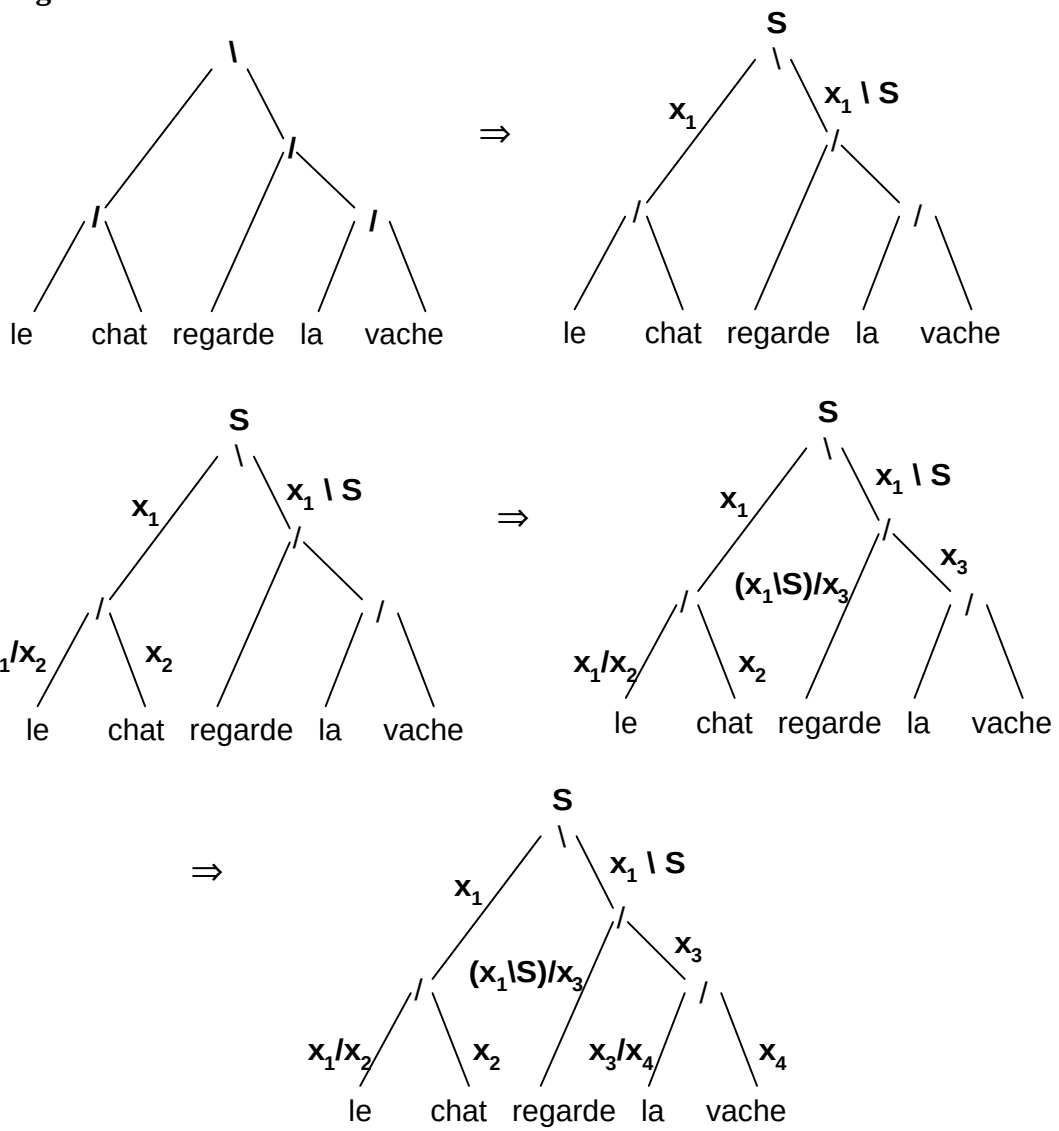
Parmi les équations entre les catégories résultant de l'identification, appliquer la règle correspondante parmi les règles ci-dessous :

(1) si A / B = C / D, alors A = C et B = D (idem pour \)
(2) si A / B = C\ D, alors échec
(3) si x = x, alors ne rien faire
(4) si t = x où *t* n'est pas un type primitif, alors remplacer par l'équation x = t
(5) si x = t et *x* a une autre occurrence dans l'ensemble des équations, alors :
si *x* apparaît dans *t*
alors échec
sinon remplacer *x* par *t* dans toutes les autres équations

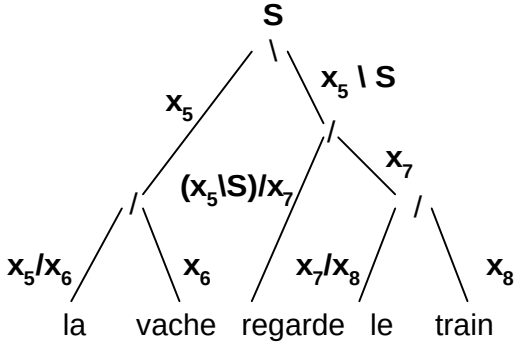
Exemple 2.3.2.6
Soit l'ensemble d'exemples positifs suivants :

{ [\ [/ le chat] [/ regarde [/ la vache]]] , [\ [/ la vache] [/ regarde [/ le train]]] }

Etiquetage :



De la même façon, on étiquette la structure d'arbre de dérivation du second exemple et on obtient l'arbre de dérivation suivant :



On obtient donc le lexique suivant :

le : x_1 / x_2 , x_7 / x_8
la : x_3 / x_4 , x_5 / x_6
chat : x_2
vache : x_4 , x_6
train : x_8
regarde : $(x_1 \setminus S) / x_3$, $(x_5 \setminus S) / x_7$

Unification :

On a plusieurs catégories pour un même mot : il faut les identifier et faire toutes les substitutions nécessaires.

- identification sur *le* : $\sigma(x_7) = x_1$ $\sigma(x_8) = x_2$
  on obtient le lexique suivant :
  { ***le*** : x_1 / x_2 ; ***la*** : x_3 / x_4 , x_5 / x_6 ; ***chat*** : x_2 ; ***vache*** : x_4 , x_6 ;
  train : x_2 ; ***regarde*** : $(x_1 \setminus S) / x_3$, $(x_5 \setminus S) / x_1$ }

- identification sur *la* : $\sigma(x_5) = x_3$ $\sigma(x_6) = x_4$
  on obtient le lexique suivant :
  { ***le*** : x_1 / x_2 ; ***la*** : x_3 / x_4 ; ***chat*** : x_2 ; ***vache*** : x_4 , x_4 ; ***train*** : x_2 ;
  regarde : $(x_1 \setminus S) / x_3$, $(x_3 \setminus S) / x_1$ }

- identification sur *vache* : $\sigma(x_4) = x_4$
  on obtient le lexique suivant :
  { ***le*** : x_1 / x_2 ; ***la*** : x_3 / x_4 ; ***chat*** : x_2 ; ***vache*** : x_4 ; ***train*** : x_2 ;
  regarde : $(x_1 \setminus S) / x_3$, $(x_3 \setminus S) / x_1$ }

- identification sur *regarde* : $\sigma(x_3) = x_1$
  on obtient le lexique suivant :
  { ***le*** : x_1 / x_2 ; ***la*** : x_1 / x_4 ; ***chat*** : x_2 ; ***vache*** : x_4 ; ***train*** : x_2 ;
  regarde : $(x_1 \setminus S) / x_1$ }

On obtient finalement la grammaire (définie par son lexique) suivante :

le : x_1 / x_2
la : x_1 / x_4
chat : x_2
vache : x_4
train : x_2
regarde : $(x_1 \setminus S) / x_1$

Il a été démontré que **l'algorithme RG converge** : Kanazawa ([5]) a montré que l'unification, qui permet de ne pas ajouter un type à chaque nouvel exemple d'utilisation d'un mot, fait converger l'algorithme.

PARTIE 3

Les grammaires de Lambek

Nous nous intéressons ici aux *grammaires de Lambek rigides*. Les grammaires de Lambek sont une extension des grammaires catégorielles : en effet, elles sont aussi définies par un lexique qui associe un type à chaque mot mais s'appuient sur un système logique de déduction : le calcul de Lambek.

Le calcul de Lambek permet ainsi de placer les grammaires catégorielles dans un formalisme mathématique plus intéressant. Ce calcul est associatif mais pour décrire les phénomènes linguistiques, on doit interdire l'associativité.

3.1 Définitions et propriétés

Pour les dérivations, on utilise les règles logiques suivantes :

- $/_e$ (élimination droite) et $\backslash_e$ (élimination gauche)
- $/_i$ (introduction droite) et $\backslash_i$ (introduction gauche) qui vont permettre notamment de gérer la montée de type

Définition 3.1.1 (Règles de dérivation).

$$\frac{}{A \vdash A}\ [ID]$$

$$\frac{\Gamma \vdash A / B \quad \Delta \vdash B}{\Gamma, \Delta \vdash A}\ [/ E] \qquad \frac{\Gamma \vdash B \quad \Delta \vdash B \backslash A}{\Gamma, \Delta \vdash A}\ [\backslash E]$$

$$\frac{\Gamma , B \vdash A}{\Gamma \vdash A / B}\ [/ I] \qquad \frac{B , \Gamma \vdash A}{\Gamma \vdash B \backslash A}\ [\backslash I]$$

Les catégories des mots appartenant au langage seront donc constituées de types primitifs et des opérateurs / et \ .

On remarque que dans le cas d'une introduction droite $/_i$ (resp. gauche $\backslash_i$), c'est l'hypothèse libre la plus à droite (resp. gauche) qui est déchargée.

La structure d'une dérivation sera donc un arbre dont les nœuds étiquetés par $/_e$ ou $\backslash_e$ seront binaires et les nœuds étiquetés par $/_i$ ou $\backslash_i$ seront unaires.

Définition 3.1.2 (Grammaire de Lambek).
Une grammaire de Lambek est un triplet (S , s, F) tel que :

- *S est un ensemble fini de symboles terminaux (le vocabulaire)*
- *s est la catégorie qui définit les phrases correctes du langage*
- *F est une fonction de S vers les sous-ensembles finis de Tp (F associe à chaque mot du vocabulaire une catégorie).*
  Si F(a) = {A_1,…, A_n}, alors on écrit G : $a \mapsto A_1,\ldots, A_n$.

Pour $w \in S^*$, $w = a_1,\ldots a_n$, on dit que G *accepte w* s'il existe une preuve dans le calcul de Lambek de $A_1,\ldots, A_n \vdash s$ avec $a_i : A_i \ \forall i$.

Définition 3.1.3 (Langage engendré).
Le langage engendré par une grammaire de Lambek (S , s, F) est l'ensemble :

$\{ a_1,\ldots a_n \in \Sigma^* \mid \forall\ 1 \le i \le n,\ \exists A_i ,\ G : a_i \mapsto A_i$ et $A_1,\ldots, A_n \vdash s \ \}$

Exemple 3.1.4.
Soient l'alphabet Σ = { la, vache, locomotive, regarde } et le lexique suivant :

la	:	NP / N
vache	:	N
locomotive	:	N
regarde	:	$(NP \backslash S) / NP$

La phrase *la vache regarde la locomotive* appartient au langage engendré par la grammaire car on peut prouver, dans le calcul de Lambek, que :

$$NP / N ,\ N ,\ (NP \backslash S) / NP ,\ NP / N ,\ N \ \vdash S$$

3.2 Apprentissage des grammaires de Lambek

3.2.1 Définitions préalables

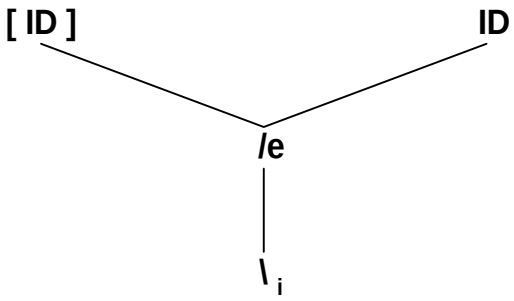
Nous ne réalisons ici que l'apprentissage de grammaires de Lambek rigides pour permettre l'unification.

Comme pour les grammaires catégorielles, l'apprentissage se fait sur des exemples positifs donnés dont on doit reconstituer les arbres de dérivation (définition **2.3.1.2**).

Définition 3.2.1.1 (Arbre de preuve).
Un arbre de preuve est un arbre dont les nœuds sont étiquetés par [ID], $/\!\!/$, $\backslash_e$, $/_i$, $\backslash_i$.

Un arbre de preuve ne contient aucune information sur la formule dont il est la preuve et sur la phrase engendrée par la grammaire utilisant cette preuve.

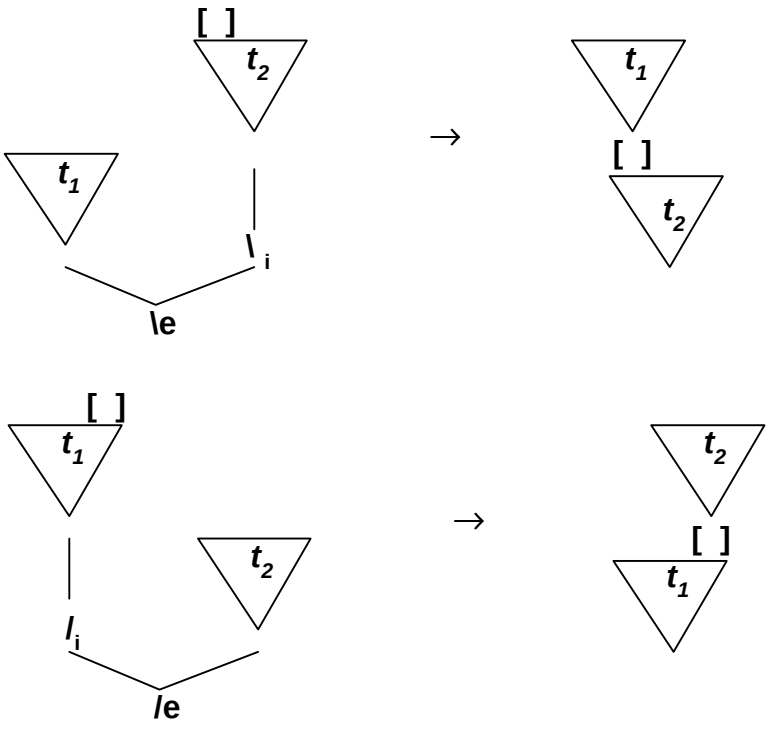
Exemple 3.2.1.2
Le terme [$\backslash_i$] ([$/_e$] ([ID] , [ID])) est un exemple d'arbre de preuve bien formé.

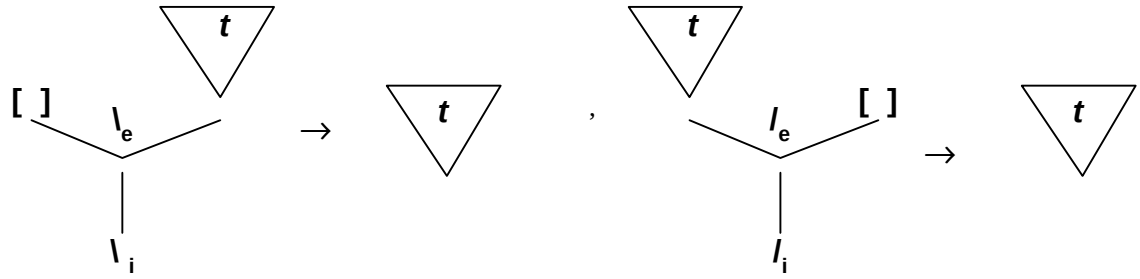


Mais il existe des exemples d'arbre de preuve mal formés :
- [$\backslash_e$] (x , [$/_i$] (y))
- [$/_e$] ([$\backslash_i$] (x) , y)
- [$\backslash_i$] ([$\backslash_e$] (x , [ID]))
- [$/_i$] ([$/_e$] ([ID] , x))

Quand un arbre de preuve est mal formé, il faut le normaliser afin de le transformer en un arbre de preuve bien formé.

Définition 3.2.1.3 (Règles de normalisation).



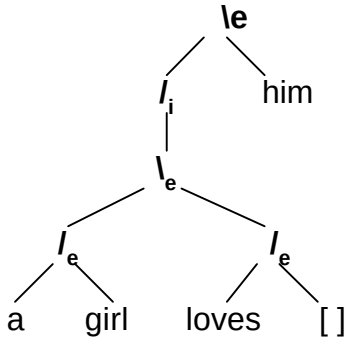


Le but de l'apprentissage est de reconstituer, pour des exemples donnés, leur arbre de dérivation indiquant les catégories des mots. Pour cela, il faut que la structure des arbres soit connue : on suppose qu'elle sera fournie avec les exemples. On définit alors les structures d'arbre de preuve.

Définition 3.2.1.4 (Structure d'arbre de preuve).
Une structure d'arbre de preuve est un arbre dont :

- *les nœuds, soit unaires soit binaires, sont étiquetés par /e ,\ e , /i ou \ i*
- *les feuilles sont étiquetées soit par* [ID] *(feuille déchargée) soit par des mots du vocabulaire*

Exemple 3.2.1.5
Soit la phrase *A girl loves him.*
Sa structure d'arbre de preuve est :



3.2.2 L'algorithme RLG

L'algorithme utilisé pour réaliser l'apprentissage est l'**algorithme RLG** qui permet d'apprendre des grammaires de Lambek rigides à partir d'exemples positifs donnés sous forme de phrases avec leur structure d'arbre de preuve.

Il comprend trois phases :

- la phase de normalisation
- la phase d'étiquetage
- la phase d'unification

Phase de normalisation :

Normaliser toutes les structures d'arbre de preuve si elles ne sont pas normales selon les règles décrites dans **3.2.1.3**.

Phase d'étiquetage :

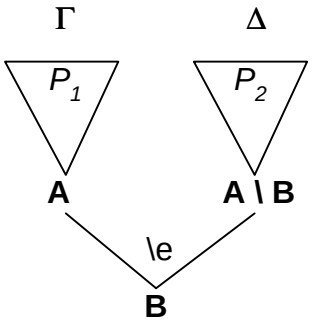
La phase d'étiquetage consiste à déduire d'une structure d'arbre de preuve un arbre de dérivation complet. Cette phase est composée de 3 étapes :

1. Affecter la catégorie **s** à la **racine** de l'arbre puisque la phrase est correcte
2. Typer les **nœuds arguments** décrits ci-dessous en **3.2.2.1** en leur affectant des catégories distinctes
3. Déduire les types des nœuds restants selon les règles données ci-dessous en **3.2.2.2**.

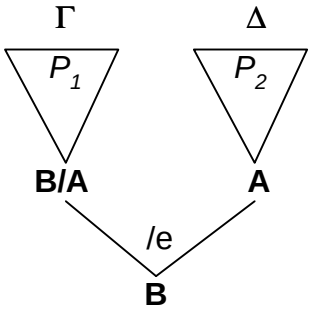
Définition 3.2.2.1 (Nœuds arguments).

Soit P un arbre partiel de dérivation en forme normale. Les nœuds arguments de R (notés Arg(P)) sont :

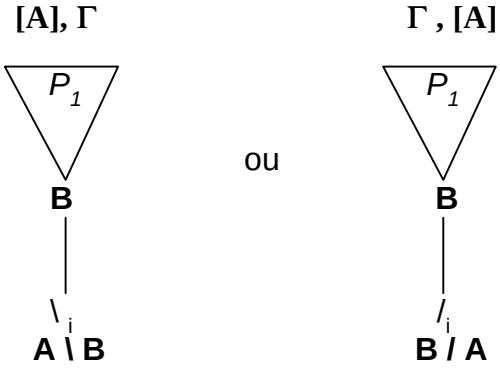
- *Arg (P) = P* si P est une feuille
- *Arg (P) = { racine(P) } ∪ Arg (P_1) ∪ Arg (P_2) – racine (P_2)* si P est de la forme :



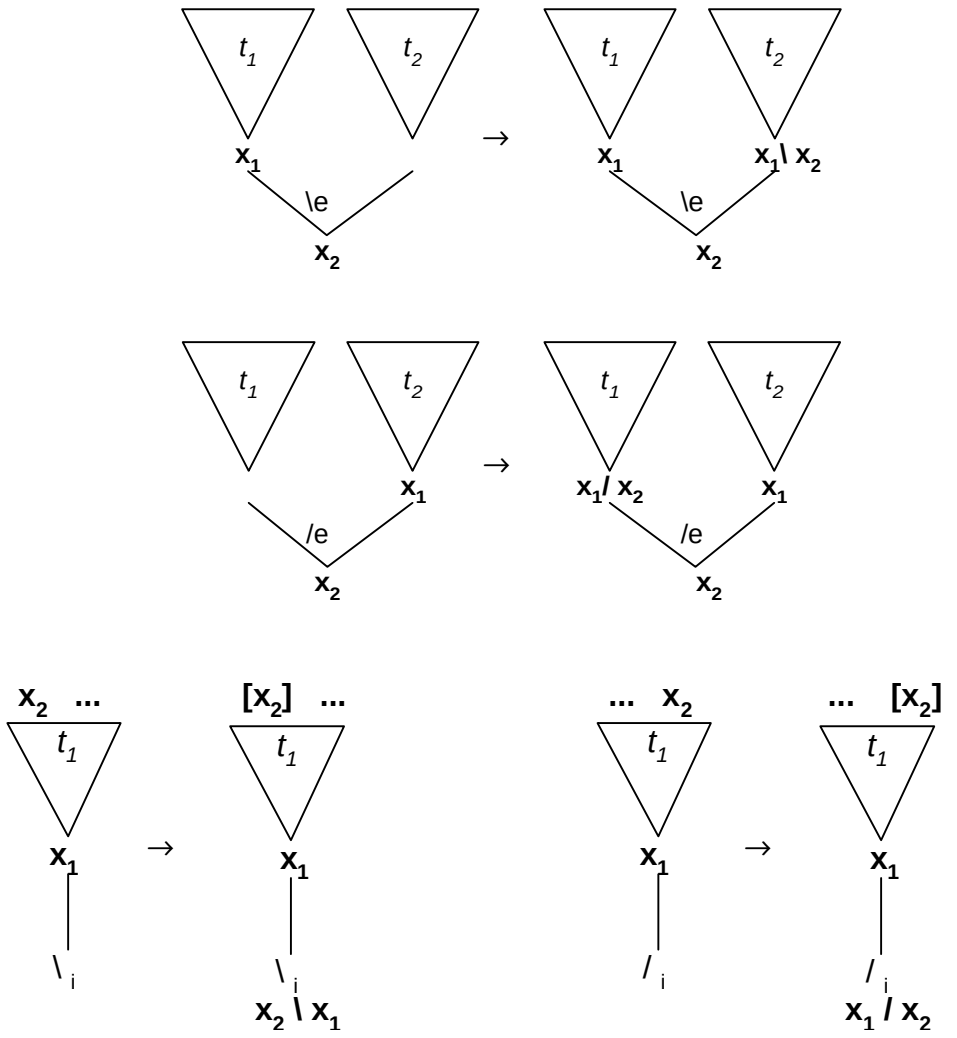
- *Arg (P) = { racine(P) } ∪ Arg (P_1) ∪ Arg (P_2) – racine (P_1)* si P est de la forme :



- *Arg (P) = P_1* si P est de la forme :



Définition 3.2.2.2 (Typage des nœuds).
Soit P une structure d'arbre de preuve en forme normale. Si tous les nœuds arguments de P sont étiquetés, alors tous les autres nœuds peuvent être étiquetés par un seul type selon les règles suivantes :



Phase d'unification :

Comme pour les grammaires catégorielles, si un mot est présent dans plusieurs exemples différents, la phase d'étiquetage précédente va lui affecter plusieurs catégories. Il faudra donc les unifier (voir **2.3.2**).

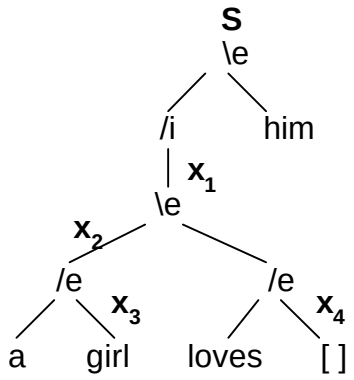
Exemple 3.2.2.3
Soit l'ensemble d'exemples positifs suivants :

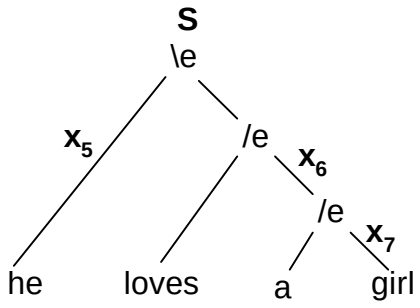
{ [\ e [/ i ([\ e [/ e a girl][/ e loves [ ]]])] him] , [\ e he [/ e loves [/ e a girl]] }

Normalisation :
Les deux arbres de preuve sont déjà en forme normale.

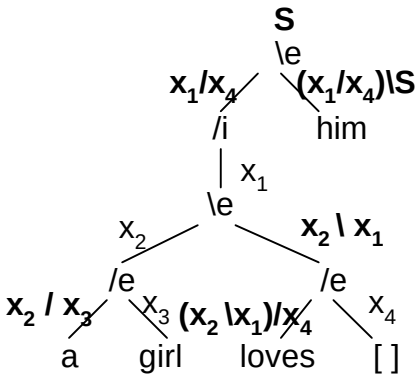
Etiquetage :

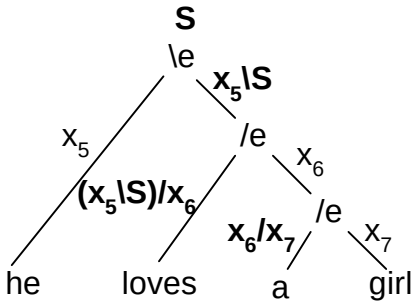
Etiquetage des nœuds arguments:





Etiquetage des autres nœuds:





On obtient donc le lexique suivant :

a	:	x_2 / x_3 , x_6 / x_7
girl	:	x_3 , x_7
he	:	x_5
loves	:	$(x_2 \backslash x_1) / x_4$, $(x_5 \backslash S) / x_6$
him	:	$(x_1 / x_4) \backslash S$

Unification :

On a plusieurs catégories pour un même mot : il faut les identifier et faire toutes les substitutions nécessaires.

- identification sur *a* : $\sigma(x_6) = x_2$ $\sigma(x_7) = x_3$
  on obtient le lexique suivant :
  { ***a*** : x_2 / x_3 ; ***girl*** : x_3 ; ***he*** : x_5 ; ***loves*** : $(x_2 \backslash x_1) / x_4$, $(x_5 \backslash S) / x_2$;
  him : $(x_1 / x_4) \backslash S$ }

- identification sur *loves* : $\sigma(x_2) = x_5$ $\sigma(x_1) = S$
  on obtient le lexique suivant :
  { ***a*** : x_5 / x_3 ; ***girl*** : x_3 ; ***he*** : x_5 ; ***loves*** : $(x_5 \setminus S) / x_5$; ***him*** : $(S / x_5) \setminus S$ }

On obtient finalement la grammaire (définie par son lexique) suivante :

a	:	x_5 / x_3
girl	:	x_3
he	:	x_5
loves	:	$(x_5 \setminus S) / x_5$
him	:	$(S / x_5) \setminus S$

Il a été démontré que **l'algorithme RLG converge** (R. Bonato, [1]).

PARTIE 4

Les grammaires catégorielles avec itérations

Les grammaires catégorielles étudiées précédemment ne proposent pas une représentation syntaxique suffisante pour représenter certains phénomènes linguistiques. Elles utilisent néanmoins la notion de dépendance syntaxique entre les mots ce qui les rapproche des grammaires de dépendance.

L'idée est d'étendre les grammaires catégorielles classiques afin non seulement de conserver leurs propriétés d'apprentissage mais aussi de prendre en compte la notion de dépendance syntaxique.

4.1 Les grammaires de dépendance

*Les mots d'un énoncé sont reliés entre eux par des **dépendances** : un mot peut ou doit dépendre d'un autre pour sa position dans la phrase et sa forme grammaticale.* (Mel'cuk, [5]).

Définition 4.1.1 (de Mel'cuk)
La dépendance est une relation non-symétrique, de même type que l'implication : l'un des éléments "présuppose" en quelque sorte l'autre, mais l'inverse n'est (en général) pas vrai. La dépendance est notée par une flèche : $w_1 \rightarrow w_2$ *signifie que* w_2 *dépend de* w_1.
w_1 *est appelé le **gouverneur** de* w_2 *(on parle aussi de la **tête** de la dépendance).*

On distingue généralement trois types de dépendances :

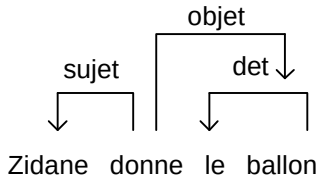
✓ **dépendances sémantiques :** la signification d'une phrase peut être représentée à l'aide du formalisme du calcul des prédicats. On dit alors que l'argument d'un prédicat dépend sémantiquement de son prédicat. Par exemple : si on considère le prédicat à trois arguments donner(Donneur, Objet, Receveur), la phrase "*Zidane donne le ballon à Dugarry*" est décrite au niveau sémantique par : *Zidane* est le donneur, *le ballon* l'objet et *Dugarry* le receveur.

✓ **dépendances morphologiques :** un mot w_1 dépend morphologiquement d'un autre mot w_2 dans une phrase donnée si au moins une des "catégories grammaticales" de w_1 est déterminée par w_2 (exemple : accord entre un nom et son déterminant).
✓ **dépendances syntaxiques :** la structure syntaxique permet de décrire l'organisation de la phrase.

Définition 4.1.2 (Arbre de dépendances).
Un arbre de dépendances est un arbre dont les nœuds sont composés des mots de la phrase, et dont les arcs expriment la dépendance d'un mot (le nœud fils) par rapport à un autre (son nœud père). Chaque arc est étiqueté par le nom de la relation syntaxique qui caractérise la dépendance.

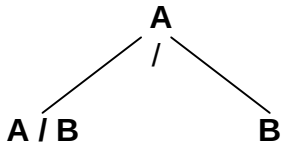
Définition 4.1.3 (Dominance).
Notons DEP(w_1,w_2) la relation qui signifie w_2 dépend de w_1, c'est-à-dire w_1 est le gouverneur de w_2. On définit alors DOM(w_1 ,w_2) par la fermeture par réflexivité et transitivité de la relation DEP. Autrement dit, w_1 domine w_2 si et seulement si w_1 est un ancêtre de w_2 dans l'arbre de dépendances.

Exemple 4.1.4



4.1.5 Grammaires de dépendance et grammaires catégorielles

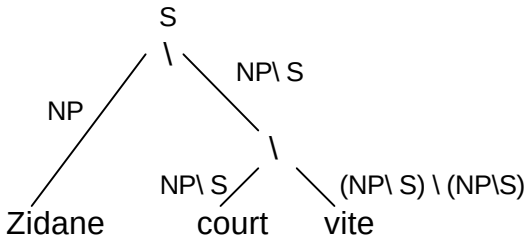
Prenons l'exemple d'une dérivation de grammaire catégorielle :



On remarque que toute dérivation est constituée de deux branches ayant chacune un rôle totalement différent : une branche argument (**B**) et une branche "principale" (**A/B**) ; on peut donc interpréter cette différence en donnant à la partie "principale" le rôle de *tête*. On peut alors construire facilement un arbre de dépendances en créant, pour chaque dérivation, une dépendance qui part de la tête de la branche "principale" vers la tête de la branche argument (et ceci récursivement jusqu'au mot).

Les grammaires catégorielles ne sont pas toujours bien adaptées pour représenter certaines dépendances. Par exemple, même si la notion de tête est sujet à débat, si l'on suppose que le nom doit être la tête du groupe nominal, le problème suivant se pose : la catégorie d'un nom est *N*, et celle d'un déterminant *NP / N*, ce qui fait du déterminant la tête du groupe nominal mais on voudrait plutôt que ce soit le nom. Il suffit alors de changer les catégories : le déterminant est alors de catégorie *D*, et le nom de catégorie *D \ NP*.

En outre, les grammaires catégorielles ne permettent pas de gérer correctement les éléments *optionnels* et les éléments *itérés*. Ces deux cas sont traités de façon identique par les grammaires catégorielles classiques, avec une catégorie de la forme C / C ou $C \backslash C$. Voici un exemple où l'adverbe est facultatif :



On remarque que les catégories font des éléments optionnels ou itératifs des têtes de dépendance. Ceci est plutôt incohérent : il n'est pas souhaitable qu'un élément facultatif soit la tête d'une dépendance.

La solution proposée par A. Dikovsky et E. Moreau ([6]) est d'étendre les grammaires catégorielles classiques aux *grammaires catégorielles avec itérations*.

4.2 Les grammaires catégorielles avec itérations

Par rapport aux grammaires catégorielles classiques, on introduit deux nouveaux opérateurs qui vont permettre de gérer les itérations (et par conséquent les optionalités).

Définition 4.2.1 (Règles de dérivation).

$$B / A \ , \ A \ \rightarrow \ B$$
$$A \ , \ A \backslash B \ \rightarrow \ B$$
$$A /^* B \ , \ B \ \rightarrow \ A /^* B$$
$$B \ , \ B \backslash^* A \ \rightarrow \ B \backslash^* A$$
$$A /^* B \ \rightarrow \ A$$
$$B \backslash^* A \ \rightarrow \ A$$

Les opérateurs / et \ ont la même signification que dans les grammaires catégorielles classiques. Pour les opérateurs /* et *, on peut dire qu'une expression de type $A/^*B$ (respectivement $B\backslash^*A$) est une expression de type **A** qui peut éventuellement être suivie (resp. précédée) d'une ou plusieurs expressions de type **B**.

Définition 4.2.2 (Grammaire catégorielle avec itérations).
Une grammaire catégorielle est un quadruplet (S , Pr, F, s) tel que :

- *S est un ensemble fini de symboles terminaux (le vocabulaire)*
- *Pr est un ensemble fini (les types primitifs)*
- *F est une fonction de S vers les sous-ensembles finis de Tp, où Tp est le plus petit ensemble tel que :*
  1. *Pr Í Tp*
  2. *Si A, B Î Tp, alors (A / B), (A \ B), (A /* B), (A * B) Î Tp*

  (F associe à chaque mot du vocabulaire une catégorie)
- *s Î Pr est la catégorie qui définit les phrases correctes du langage*

Définition 4.2.3 (Langage engendré).
Le langage engendré par une grammaire catégorielle avec itérations (S, Pr, F, s) est l'ensemble :

$$\{ a_1,\ldots,a_n \in \Sigma^* \mid \forall\ 1 \le i \le n,\ \exists A_i : F(a_i) = A_i \ \text{ et } A_1,\ldots, A_n \ \Re^*\ s \}$$

avec $\Re^*$ *la fermeture transitive et réflexive de* $\Re$.

Exemple 4.2.4
Soit la grammaire suivante : { le : *D* ; joli, petit : *A* ; chien : *A*(D \ NP)* ; dort : *NP \ S* }

Ci-dessous la dérivation de la phrase *le joli petit chien dort* :

D, A, A, A (D \ NP), NP \ S* $\Re$
D, A, A (D \ NP), NP \ S* $\Re$
D, A (D \ NP), NP \ S* $\Re$
D, D \ NP, NP \ S $\Re$
NP, NP \ S $\Re$
S

4.3 Apprentissage des grammaires catégorielles avec itérations

4.3.1 Définitions préalables

Nous ne réalisons ici que l'apprentissage de grammaires catégorielles rigides avec itérations pour permettre l'unification.

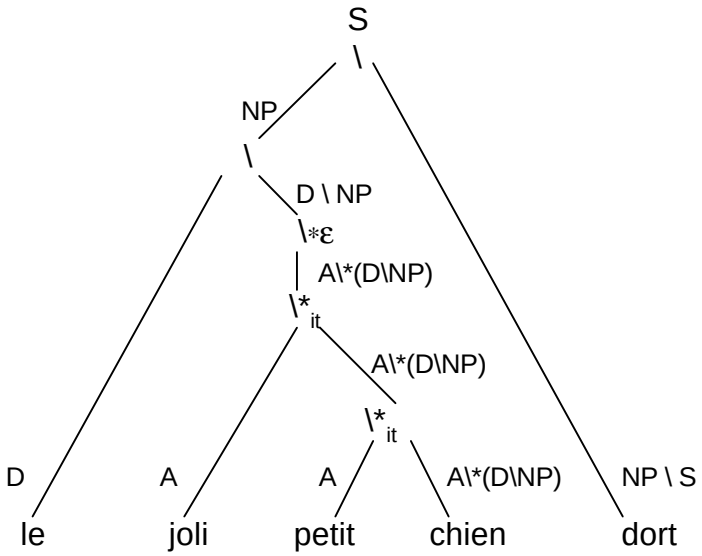
Comme pour les grammaires catégorielles classiques, l'apprentissage se fait sur des exemples positifs donnés (phrases bien formées) dont on doit reconstituer les arbres de dérivation. Un arbre de dérivation pour une grammaire catégorielle avec itérations (Σ, Pr, F, s) est défini de la manière suivante :

Définition 4.3.1.1 (Arbre de dérivation).
Si D est une dérivation de B à partir de $A_1,\ldots, A_n$ *et* $a_1,\ldots,a_n$ *Î S tels que* $G : a_i \mapsto A_i$ *pour 1 £ i £ n, l'arbre constitué par D aux feuilles duquel sont attachés les* $a_1,\ldots,a_n$ *de gauche à droite dans cet ordre, est un arbre de dérivation partiel de G.*

Un arbre de dérivation est complet si la catégorie qui figure à sa racine est s.

Les nœuds d'un arbre de dérivation sont marqués par un opérateur qui désigne la règle de réduction utilisée :
- Opérateurs binaires : / , \ , $/*_{it}$, $\backslash*_{it}$
  - $B / A\ ,\ A \Rightarrow_{/} B$
  - $A\ ,\ A \backslash B \Rightarrow_{\backslash} B$
  - $A /* B\ ,\ B \Rightarrow_{/*it} A /* B$
  - $B\ ,\ B \backslash* A \Rightarrow_{\backslash*it} B \backslash* A$
- Opérateurs unaires : $/*_{\varepsilon}$, $\backslash*_{\varepsilon}$ (qui indiquent quand l'itération se termine)
  - $A /* B \Rightarrow_{/*\varepsilon} A$
  - $B \backslash* A \Rightarrow_{\backslash*\varepsilon} A$

Exemple 4.3.1.2
Soit la phrase *le joli petit chien dort*.
Son arbre de dérivation est :



Le but de l'apprentissage est de reconstituer, pour des exemples donnés, leur arbre de dérivation indiquant les catégories des mots. Pour cela, il faut que la structure des arbres soit connue : on suppose qu'elle sera fournie avec les exemples.

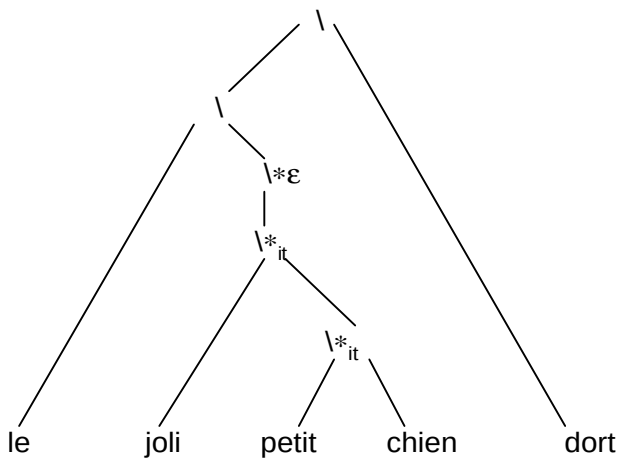
Définition 4.3.1.3 (Structure d'arbre de dérivation).
Une structure d'arbre de dérivation est un arbre de dérivation pour lequel les catégories des nœuds et des feuilles ne sont pas précisées.

La structure d'arbre de dérivation doit donc comprendre, pour chaque nœud, l'opérateur utilisé (/ , \ , $/*_{it}$, $\backslash*_{it}$, $/*_{\varepsilon}$, $\backslash*_{\varepsilon}$).

Exemple 4.3.1.4
Reprenons la phrase *le joli petit chien dort.*
Sa structure d'arbre de dérivation peut être représentée de deux manières :

$[_{\backslash}$ $[_{\backslash}$ le $[_{\backslash*\varepsilon}$ $[_{\backslash*it}$ joli $[_{\backslash*it}$ petit chien]]]] dort]

ou



4.3.2 L'algorithme RG*

L'algorithme utilisé pour réaliser l'apprentissage des grammaires catégorielles avec itérations est l'**algorithme RG*** : il s'inspire de l'algorithme RG étudié précédemment. Il permet lui aussi d'apprendre des grammaires catégorielles rigides avec itérations à partir d'exemples positifs donnés sous forme de phrases avec leur structure d'arbre de dérivation.

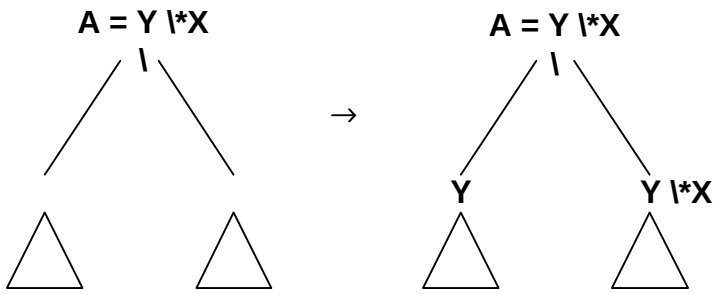
Il comprend deux phases :

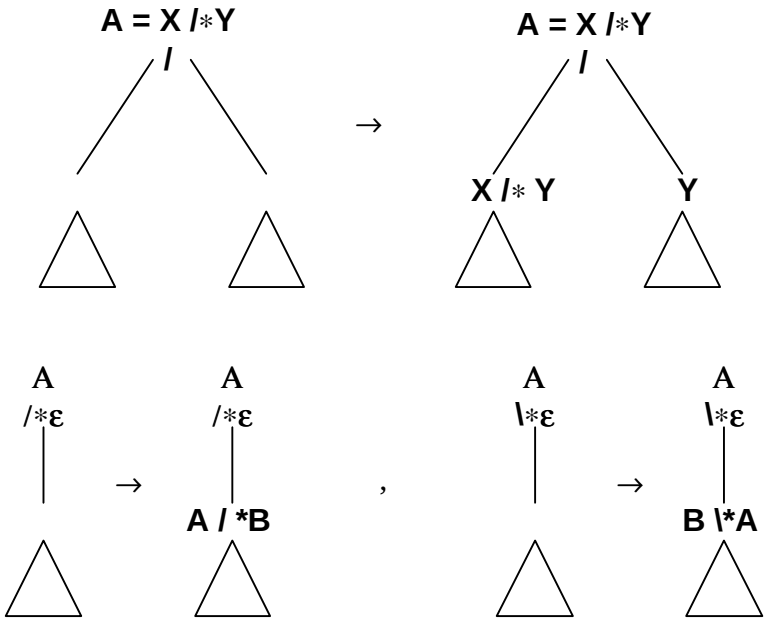
- la phase d'étiquetage
- la phase d'unification

Phase d'étiquetage :

La phase d'étiquetage consiste à déduire d'une structure d'arbre de dérivation un arbre de dérivation complet, c'est-à-dire à déduire les catégories des mots en connaissant les règles utilisées. Pour cela, on procède de la façon suivante :

1. Affecter la catégorie **s** à la racine de l'arbre puisque la phrase est correcte puis étiqueter les sous-arbres de la manière suivante :
2. Etiquetage d'un nœud de catégorie **A** :
   - si l'opérateur est / :
     1. affecter au nœud de l'opérande droit une nouvelle catégorie **B**
     2. affecter au nœud de l'opérande gauche la catégorie **A / B**
     3. étiqueter les sous-arbres
   - si l'opérateur est \ :
     1. affecter au nœud de l'opérande gauche une nouvelle catégorie **B**
     2. affecter au nœud de l'opérande droit la catégorie **B \ A**
     3. étiqueter les sous-arbres
   - si l'opérateur est $/^*_{it}$:
     1. la catégorie *A* est de type $A = X /^* Y$. Affecter au nœud de l'opérande gauche la catégorie $\mathbf{A = X /^* Y}$
     2. affecter au nœud de l'opérande droit la catégorie **Y**
     3. étiqueter les sous-arbres
   - si l'opérateur est $\backslash^*_{it}$:
     1. la catégorie *A* est de type $A = Y \backslash^* X$. Affecter au nœud de l'opérande droit la catégorie $\mathbf{A = Y \backslash^* X}$
     2. affecter au nœud de l'opérande gauche la catégorie **Y**
     3. étiqueter les sous-arbres
   - si l'opérateur est $/^*_{\varepsilon}$ (resp. $\backslash^*_{\varepsilon}$) :
     1. affecter à la racine du sous-arbre la catégorie $\mathbf{A/^*B}$ (resp. $\mathbf{B \backslash^* A}$)
     2. étiqueter les sous-arbres





Phase d'unification :

Comme pour les grammaires catégorielles classiques, si un mot est présent dans plusieurs exemples différents, la phase d'étiquetage précédente va lui affecter plusieurs catégories. Il faudra donc les unifier (voir **2.3.2**).

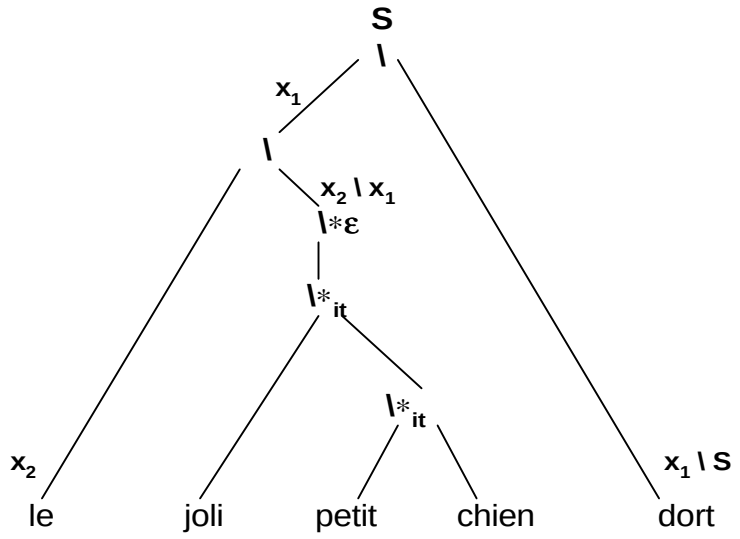
Exemple 4.3.2.1

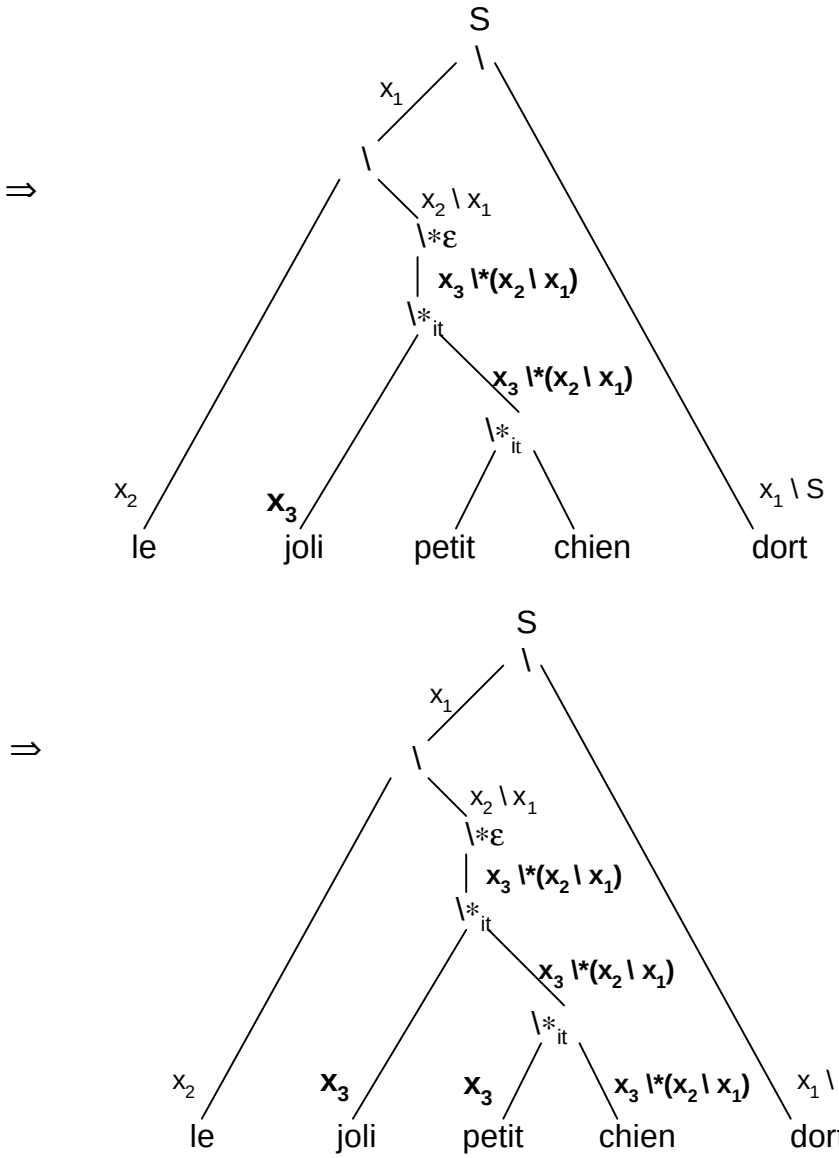
Soit l'ensemble d'exemples positifs suivants :

{ [\ [\ le [*ε [*it joli [*it petit chien]]]] dort] ,

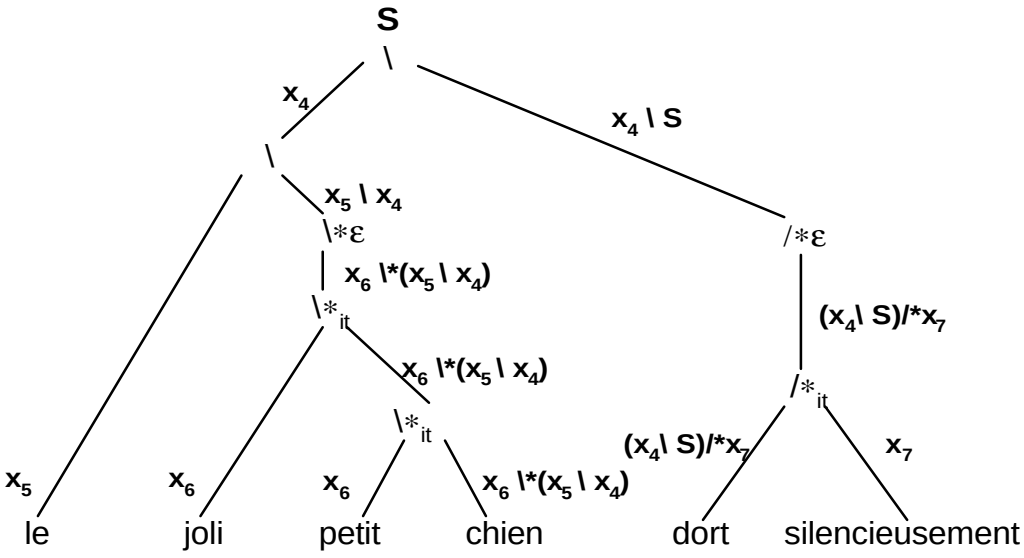
[\ [\ le [*ε [*it joli [*it petit chien]]]] [/*ε [/*it dort silencieusement]]] }

Etiquetage :





De la même façon, on étiquette la structure d'arbre de dérivation du second exemple et on obtient l'arbre de dérivation suivant :



On obtient donc le lexique suivant :

le : x_2 , x_5
joli : x_3 , x_6
petit : x_3 , x_6
chien : $x_3 \backslash^*(x_2 \backslash x_1)$, $x_6 \backslash^*(x_5 \backslash x_4)$
dort : $x_1 \backslash S$, $(x_4 \backslash S) /^* x_7$
silencieusement : x_7

Unification :

On a plusieurs catégories pour un même mot : il faut les identifier et faire toutes les substitutions nécessaires.

- identification sur *le* : $\sigma(x_5) = x_2$
  on obtient le lexique suivant :
  { ***le*** : x_2 ; ***joli*** : x_3 , x_6 ; ***petit*** : x_3 , x_6 ;
  chien : $x_3 \backslash^*(x_2 \backslash x_1)$, $x_6 \backslash^*(x_2 \backslash x_4)$; ***dort*** : $x_1 \backslash S$, $(x_4 \backslash S) /^* x_7$;
  silencieusement : x_7 }

- identification sur *joli* : $\sigma(x_6) = x_3$
  on obtient le lexique suivant :
  { ***le*** : x_2 ; ***joli*** : x_3 ; ***petit*** : x_3 ; ***chien*** : $x_3 \backslash^*(x_2 \backslash x_1)$, $x_3 \backslash^*(x_2 \backslash x_4)$;
  dort : $x_1 \backslash S$, $(x_4 \backslash S) /^* x_7$; ***silencieusement*** : x_7 }

- identification sur *chien* : $\sigma(x_4) = x_1$
  on obtient le lexique suivant :
  { ***le*** : x_2 ; ***joli*** : x_3 ; ***petit*** : x_3 ; ***chien*** : $x_3 \backslash^*(x_2 \backslash x_1)$;
  dort : $x_1 \backslash S$, $(x_1 \backslash S) /^* x_7$; ***silencieusement*** : x_7 }

- identification sur *dort* : $\sigma(x_1\backslash S) = (x_1\backslash S)/^* x_7$
  on obtient le lexique suivant :
  { ***le*** : x_2 ; ***joli*** : x_3 ; ***petit*** : x_3 ; ***chien*** : $x_3 \backslash^*(x_2 \backslash x_1)$;
  dort : $(x_1 \backslash S) /^* x_7$; ***silencieusement*** : x_7 }

On obtient finalement la grammaire (définie par son lexique) suivante :

le : x_2
joli : x_3
petit : x_3
chien : $x_3 \backslash^*(x_2 \backslash x_1)$
dort : $(x_4 \backslash S) /^* x_7$
silencieusement : x_7

Il a été démontré que **l'algorithme RG* converge** (E. Moreau, [6]).

PARTIE 5

Détails d'implémentation

Notre travail a consisté à implémenter les différents algorithmes d’apprentissage et à les intégrer dans une interface. Outre la classique et nécessaire présentation de notre travail, cette partie se veut aussi être un guide le plus pratique possible pour la continuation du développement de l'application par d'autres informaticiens.

5.1 Contenu des fichiers

5.1.1 Langages et librairies utilisés

Les algorithmes sont implémentés en Objective CaML. Les exemples d’arbres sont donnés en XML. Une interface graphique en Tcl/Tk permet de visualiser les arbres.

Le code CaML suivant correspond à l’ouverture et au chargement des bibliothèques utilisées, elles peuvent varier suivant la configuration du système sur lequel on exécute l’environnement :

```
open Tk;;
open Graphics;;
open Pxp_yacc;;
open Pxp_document;;
open List;;
open String;;
open Str;;

#load "str.cma";;
#load "graphics.cma";;
#load "unix.cma";;
#load "labltk.cma";;
#load "libjpf.cma";;
#load "/usr/local/lib/ocaml/site-lib/netstring/netstring.cma";;
#load "pxp_engine.cma";;
#load "pxp_lex_utf8.cma";;
#load "pxp_lex_iso88591.cma";;
#load "pxp_lex_link_iso88591.cmo";;
```

Par rapport à la version précédente, seules les librairies `List`, `String` et `Str` ont été ajoutées. Elles permettent l’utilisation des fonctions sur les listes et les chaînes de caractères.

5.1.2 Organisation des fichiers

Le fichier "header" contient toutes les références aux librairies utilisées. En outre, il ouvre dans le bon ordre les autres fichiers du programme : "sortie.ml", "algorithmes.ml" et "interface.ml".

Le fichier "sortie.ml" contient les fonctions nécessaires à la conversion des arbres XML en arbres CaML et à l'affichage sous forme graphique des arbres (utilise Tcl/Tk). Le fichier "algorithmes.ml" contient l'implémentation des trois algorithmes (RG, RG* et RLG) ainsi que l'implémentation de l'unification. Enfin, le fichier "interface.ml" (ancien fichier "stage") contient une version complètement réécrite mais encore incomplète et imparfaite de l'interface du programme.

5.2 Représentation des arbres

5.2.1 Structure des arbres en XML

La structure des arbres en XML correspond à la définition XML suivante :

```
<!ELEMENT DOCUMENT (DAG)+>
<!ELEMENT DAG (NOEUD)>
<!ELEMENT NOEUD (ETIQ,(ETIQ?,NOEUD)*)>
<!ELEMENT ETIQ (#PCDATA)>
```

Pour expliciter cette structure, nous allons étudier cette définition ligne par ligne :

<!ELEMENT DOCUMENT (DAG)+> Cette ligne implique deux règles : la racine d'un fichier XML est la balise <DOCUMENT> , et cette balise peut contenir un ou plusieurs éléments <DAG>, ce qui signifie qu'un fichier XML conforme à la définition précédente pourra contenir un ou plusieurs arbres.

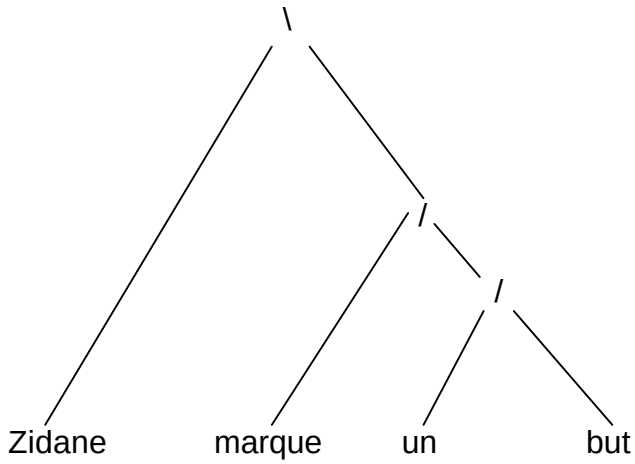
<!ELEMENT DAG (NOEUD)> Une balise <DAG> contient une et une seule balise <NOEUD>. Cela correspond au fait qu'un arbre est défini comme une imbrication de nœuds. Comme un arbre n'a qu'une racine, la balise <DAG> qui délimite l'arbre ne peut effectivement accueillir qu'un nœud.

<!ELEMENT NOEUD (ETIQ,(ETIQ?,NOEUD)*)> Cette balise signifie que l'élément <NOEUD> contient une (et une seule) balise <ETIQ> suivie de zéro, une ou plusieurs fois la séquence constituée d'une balise <ETIQ> (facultative) et d'une balise <NOEUD>.

Ainsi, le contenu de cette balise est récursif : un nœud peut posséder zéro, un ou deux fils (dans cette structure, on peut en accepter plus, ce qui est incorrect). L'utilisation de la récursivité permet d'implémenter la notion d'imbrication de nœuds. La première balise <ETIQ> (non facultative) impose de donner une règle pour la racine de l'arbre ou un mot pour les feuilles. L'autre balise <ETIQ> (optionnelle) permet d'associer un type aux arêtes.

<!ELEMENT ETIQ (#PCDATA)> Les éléments <ETIQ> contiennent du texte. Ce texte désigne : soit un mot, soit un type de règle, soit une catégorie. Le niveau d'imbrication des nœuds donne le niveau du nœud dans l'arbre.

Par exemple, l'arbre suivant :



peut se représenter, en suivant la définition pré-citée, en XML par :

```
<DAG>
  <NOEUD>
   <ETIQ>elimg</ETIQ>
    <NOEUD>
     <ETIQ>Zidane</ETIQ>
    </NOEUD>
    <NOEUD>
     <ETIQ>elimd</ETIQ>
      <NOEUD>
       <ETIQ>marque</ETIQ>
      </NOEUD>
      <NOEUD>
       <ETIQ>elimd</ETIQ>
        <NOEUD>
         <ETIQ>un</ETIQ>
        </NOEUD>
        <NOEUD>
         <ETIQ>but</ETIQ>
        </NOEUD>
      </NOEUD>
    </NOEUD>
  </NOEUD>
</DAG>
```

5.2.2 Conversion des arbres XML en CaML

L'interface prévoit la conversion d'arbres XML en arbres CaML. Il suffit pour cela d'utiliser les lignes CaML suivantes :

```
let doc = parse_document_entity default_config (from_file "DAG.xml") spec ;;
let liste_arbre = creer_arbre doc ;;
```

On peut remplacer le fichier `DAG.xml` par n'importe quel fichier XML bien formé contenant le ou les arbres que l'on souhaite convertir. Les arbres sont alors contenus dans la liste d'éléments de type '`arbre`' `liste_arbre`.

5.2.3 Structure des arbres en CaML

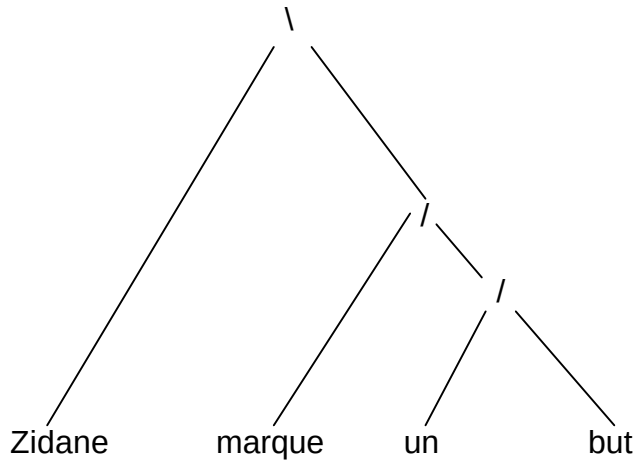
En CaML, le type `arbre` est défini de la façon suivante, conjointement au type `arete` :

```
type 'a arbre = Noeud of ('a * (('a arete) list))
and  'a arete = Arete of ('a * ('a arbre)) ;;
```

Un arbre est alors représenté par une imbrication de nœuds. Le premier nœud désigne la racine de l'arbre. La profondeur dans l'imbrication renseigne sur la profondeur dans l'arbre. Un nœud est constitué d'un couple (étiquette, liste d'arêtes). Une arête étant elle-même un couple (étiquette correspondant à la catégorie du nœud qui suit, arbre). Ainsi la liste d'arêtes d'un nœud pourra :

- être vide : le nœud est une feuille,
- contenir une arête (utile dans les grammaires de Lambek pour les règles d'introduction $/_i$ et $\backslash_i$),
- contenir deux arêtes (les deux fils du nœud courant).

Par exemple, l'arbre suivant :



sera représenté en CaML par :

```
Noeud("elimg",
     [ Arete("",Noeud("Zidane",[]));
       Arete("",Noeud("elimd",[ Arete("",Noeud("Marque",[]));
                                Arete("",Noeud("elimd",[ Arete("",Noeud("un",[]));
                                                         Arete("",Noeud("but",[]))
                                                       ]
                                       )
                                )
                       ]
             )
      )
     ]
    )
```

5.3 Représentation du lexique

5.3.1 Représentation du lexique en CaML

En CaML, le lexique est défini de la façon suivante :

```
type 'a lexeme = Mot of ('a * 'a );;
type 'a lexique = (('a lexeme) list);;
```

Le lexique est donc une liste de `Mot`. Un `Mot` étant un couple (Mot, Type). Par exemple, le lexique :

```
Zidane, NP
marque, NP\S/NP
un, NP/N
but, N
```

sera représenté par :

```
[ Mot("Zidane", "NP"); Mot("marque", "NP\S/NP"); Mot("un", "NP/N");
  Mot("but", "N") ]
```

5.3.2 Persistance du lexique

L'application de l'algorithme d'unification créé un lexique. Le lexique est affiché dans la console en mode texte. Sa persistance est prévue sous forme de fichier texte. Par exemple, le lexique suivant :

```
[ Mot("Zidane", "NP"); Mot("marque", "NP\S/NP"); Mot("un", "NP/N");
  Mot("but", "N") ]
```

sera enregistré sous forme de fichier texte au format suivant :

```
Zidane : NP
marque : NP\S/NP
un : NP/N
but : N
```

Une fonctionnalité possible de l'environnement serait d'enregistrer le lexique dans le format XML déjà défini. Nous n'avons pas implémenté cette possibilité.

5.4 Représentation des règles et des catégories

5.4.1 Convention de représentation des règles

Dans les étiquettes des arbres XML, on a adopté les conventions suivantes :

- `"elimg"` pour la règle `'\'`,
- `"elimd"` pour la règle `'/'`,
- `"itg"` pour la règle `'\*'`,

- "itd" pour la règle '/*',
- "epsg" pour la règle '\ε',
- "epsd" pour la règle '/ε',
- "introg" pour la règle '\$_i$',
- "introd" pour la règle '/$_i$'.

5.4.2 Représentation des catégories en CaML

Dans l'arbre comme dans le lexique, les catégories sont représentées par des chaînes de caractères dont le format est bien spécifique :
- la règle *élimination droite* est représentée par le caractère '/',
- la règle *élimination gauche* est représentée par le caractère '|',
- la règle *élimination droite avec itérations* est représentée par la chaine '/*',
- la règle *élimination gauche avec itération*s est représentée par la chaine '|*'.

On a utilisé le '|' en lieu et place du '\' parce qu'il ne semble pas possible en CaML d'utiliser ce caractère spécial même en utilisant les astuces habituelles ('\\', utilisation du code ascii du caractère, etc…).

En outre, les types ne doivent pas comporter d'espace. En effet, les fonctions que nous avons implémentées qui travaillent sur des chaînes de caractères n'ignorent pas les espaces.

Enfin, si le calcul de Lambek autorise l'associativité, en linguistique, celle-ci n'est pas toujours souhaitable. Par exemple, le lexique associé à la phrase "Zidane court vite" est le suivant :

```
Zidane, NP
court, NP\S
vite, (NP\S)\(NP\S)
```

On ne peut pas modifier le parenthèsage de l'adverbe vite sans quoi on ne peut plus induire l'axiome S. C'est pourquoi, dans l'apprentissage, à chaque fois qu'on applique l'étiquetage à un sous-arbre on parenthèse le type de la racine (passé en paramètre).

5.5 Quelques explications sur l'implémentation des différents algorithmes

5.5.1 Algorithmes d'apprentissage

Il n'est pas difficile de comprendre l'implémentation des algorithmes RG, RG* et RLG car celle-ci correspond assez directement à ce qui a été expliqué dans les parties précédentes. Ce que l'on peut ajouter ici, c'est que la récursivité a largement été employée pour deux raisons :
- cela correspond à l'esprit de CaML,
- la représentation sous forme d'arbre se prête bien à l'utilisation d'algorithmes récursifs.

On peut en outre présenter ici la méthode utilisée pour numéroter les types lors de l'étiquetage. En effet, il faut à la fois pouvoir utiliser le même numéro dans deux branches filles d'une branche racine tout en utilisant des numéros différents dans l'exploration des deux sous-arbres (qui se fait parallèlement du fait de l'emploi de la récursivité). De même, il faut utiliser des numéros différents pour chaque nouvel arbre étiqueté. C'est pourquoi on emploie

une variable globale (on utilise ici les fonctionnalités objet d'Objective CaML). Voici la classe implémentée à cet effet :

```
class compteur = object
  val mutable cpt = 0            (* initialisation du compteur à 0 *)
  method get = cpt               (* val courante du compteur       *)
  method reset = cpt <- 0        (* remise à 0 du compteur         *)
  method incr = cpt <- cpt+1     (* incrémentation du compteur     *)
  method nothing = cpt <- cpt    (* ne fait rien !                 *)
end ;;
```

5.5.2 Présentation de notre implémentation de la normalisation

L'étape de normalisation consiste à convertir des arbres de preuve en forme normale s'ils ne le sont pas déjà.

C'est la fonction récursive `normalisation` qui réalise cette transformation : elle prend en entrée un arbre de preuve et renvoie l'arbre en forme normale.

Cette fonction parcourt l'arbre de preuve : dans le cas d'arbres ou de sous-arbres mal formés, il y a deux possibilités :

- soit un nœud $/_i$ (resp. $\backslash_i$) est suivi d'un nœud $/_e$ (resp. $\backslash_e$), dans ce cas on renvoie le sous-arbre gauche (resp. droit)
- dans les autres cas d'arbres mal formés, il faut faire les modifications nécessaires grâce à la fonction `remplace_arbre`.

Cette dernière prend en paramètre deux sous-arbres qu'il faut relier entre eux et le type d'introduction du nœud mal formé. Pour cela, elle va se positionner à l'endroit de la feuille déchargée du premier sous-arbre (on se déplace soit dans le sous-arbre gauche soit dans le sous-arbre droit en fonction du type de la règle d'introduction) et la remplace par le second sous-arbre.

5.5.3 Présentation de notre implémentation de l'unification

L'unification se fait en deux étapes :

(i) la création du lexique, qui consiste simplement à récupérer les types de chaque mot (types obtenus par application des algorithmes d'apprentissage) et à produire une liste de couples (mot, catégorie). Nous ne développerons pas cette partie qui ne pose aucun problème.

(ii) la seconde étape est l'unification à proprement parler qui se décompose elle-même en trois phases : identification, décomposition et substitution. L'algorithme ayant été explicité dans les parties précédentes, nous nous attacherons uniquement à présenter les subtilités de l'implémentation.

L'identification consiste à trouver les mots pour lesquels il existe deux catégories différentes. Cette partie de l'unification ne pose pas de problème. Une fois qu'on a identifié ces deux types, on doit les décomposer. L'utilisation des chaînes de caractères pour la représentation des types impose alors quelques manipulations non triviales. La décomposition se fait suivant l'opérateur principal. Il existe plusieurs règles pour trouver l'opérateur principal :

- soit le type ne contient pas de parenthèse et alors il contient au plus un opérateur : dans ce cas, on décompose la catégorie par rapport à celui-ci,

- soit le type contient des parenthèses et alors il possède au moins deux opérateurs : par exemple, le type `"((A/B)/(C/D))/(E/F)"` se décompose en `"(A/B)/(C/D)"` et `"(E/F)"`. Il apparaît alors qu'on peut trouver l'opérateur principal en s'aidant du parenthèsage. L'opérateur principal est le premier opérateur qui apparaît immédiatement derrière la première expression bien parenthésée.

C'est le rôle des fonctions `position_operateur_principal`, qui renvoie la position de l'opérateur principal et `compte_parenthese`, qui donne la position qui suit la fin de la première expression bien parenthésée.

Le parenthésage des expressions données en entrée de la fonction `position_operateur_principal` doit être correct. C'est bien le cas puisque, dans notre implémentation, le parenthésage n'est pas l'œuvre d'un utilisateur mais des algorithmes d'étiquetage. Dans ce contexte, trouver le premier opérateur suivant une expression bien parenthésée ne pose pas le problème de vérifier si l'expression est bien parenthésée. On se contente ici de compter les parenthèses ouvrantes et fermantes. Lorsqu'on a trouvé autant de parenthèses fermantes que de parenthèses ouvrantes, on a fini d'explorer la partie de l'expression située à gauche de l'opérateur principal (et ainsi, on a trouvé la position de l'opérateur principal). C'est pourquoi nous n'utilisons pas d'algorithme standard pour l'analyse des expressions bien parenthésées mais un algorithme dédié.

Connaissant la position de l'opérateur principal, la décomposition ne pose pas de problème : la fonction `decomp_en_deux` construit une liste de sous-catégories en décomposant la catégorie passée en paramètre autour de l'opérateur principal puis, récursivement, la fonction `decompose_type` applique cette décomposition aux sous-catégories ainsi construites. Au passage, il faut veiller à supprimer les parenthèses qui englobent ces sous-catégories : c'est le rôle de la fonction `enleve_parentheses`.

On applique cette décomposition au type à remplacer ainsi qu'au type remplaçant. Ce double travail est commandé par la fonction `unif`.

Maintenant qu'on a identifié les types à substituer et leurs remplaçants, il faut réaliser la substitution dans tout le lexique. C'est ce qu'effectue la fonction `substitution` en utilisant la fonction `substituer_type` pour réaliser la substitution concrète d'un type par un autre. Les fonction `unif` et `uni` permettent de réaliser la substitution sur l'ensemble des types décomposés. Enfin, la fonction `unification` réitère ce processus jusqu'à ce qu'il n'y ait plus de doublons.

5.5.4 Complexité

Les trois algorithmes d'apprentissage RG, RG* et RLG ont une complexité en $o(n)$, où n représente le nombre de nœuds : on applique récursivement le même traitement à chaque nœud en partant de la racine jusqu'aux feuilles.

L'unification est en $o(n^5)$, où n est la taille du lexique non unifié. En réalité, la complexité dépendra du nombre de mots identiques à chaque étape : comme dans le cas courant peu de mots identiques apparaissent à une étape donnée et que l'unification est appliquée sur un lexique unifié et un unique nouvel exemple, seuls q mots sur les n mots du lexique auront un traitement en $o(n^5)$.

Le coût total de l'apprentissage est donc aussi en $o(n^5)$ puisqu'on applique l'unification p fois, où p désigne le nombre d'exemples.

5.5.5 Exemple d'exécution pas à pas

On veut appliquer l'algorithme RG sur les exemples positifs suivants :

le chat regarde la vache et *la vache regarde le train*

- Ces deux phrases sont données sous forme d'arbre XML et sont convertis en CaML. On obtient ainsi la liste d'arbre suivante :

```
[ Noeud("elimg",
   [ Arete("",Noeud("elimd",[ Arete("",Noeud("le",[]));
                              Arete("",Noeud("chat",[]))]));
     Arete("",Noeud("elimd",[ Arete("",Noeud("regarde",[]));
                              Arete("",Noeud("elimd",[ Arete("",Noeud("la",[]));
                                                       Arete("",Noeud("vache",[]))
                                                     ]
                                                )
                                     )
                            ]
                  )
         )
   ]
    ) ;
Noeud("elimg",
   [ Arete("",Noeud("elimd",[ Arete("",Noeud("la",[]));
                              Arete("",Noeud("vache",[]))]));
     Arete("",Noeud("elimd",[ Arete("",Noeud("regarde",[]));
                              Arete("",Noeud("elimd",[ Arete("",Noeud("le",[]));
                                                       Arete("",Noeud("train",[]))
                                                     ]
                                                )
                                     )
                            ]
                  )
         )
   ]
    )
]
```

- On appelle alors la fonction `grammaireAB` sur cette liste d'arbres. Comme la liste contient deux arbres, on fait l'opération :

```
unification( creer_lexique( algoRG(hd(listarb),"S",c#incr),"",[])
             @ grammaireAB(tl(listarb)) )
```

puis, comme il n'y a plus qu'un arbre dans la liste, on a :

```
grammaireAB(tl(listarb)) =
     unification(creer_lexique(algoRG(hd(tl(listarb)),"S",c#reset),"",[]))
```

Finalement, on obtient :

```
unification(
        creer_lexique( algoRG(hd(listarb),"S",c#incr),"",[])
        @ unification(
                   creer_lexique(algoRG(hd(tl(listarb)),"S",c#reset),"",[])
                    )
           )
```

- On commence donc par appliquer l'algorithme RG (fonction `algoRG`) sur le second arbre de la liste en lui passant également en paramètre le type de la racine (`"S"`) ainsi qu'un compteur pour numéroter les catégories. On obtient alors l'arbre suivant :

```
Noeud("elimg",
 [Arete("0",Noeud("elimd",[Arete("0/1",Noeud("la",[]));
                           Arete("1",Noeud("vache",[]))]));
  Arete("0\S",Noeud("elimd",[ Arete("(0\S)/2",Noeud("regarde",[]));
                              Arete("2",Noeud("elimd",[Arete("2/3",Noeud("le",[]));
                                                       Arete("3",Noeud("train",[]))
                                                      ]
                                                )
                                       )
                              ]
                    )
       )
 ]  )
```

NB : on affecte les catégories aux arêtes.

- On applique ensuite la fonction `creer_lexique` sur cet arbre en lui passant également en paramètre une chaîne de caractères vide au départ (dans les appels récursifs, ce sera l'étiquette correspondant au nœud étudié) ainsi qu'une liste correspondant au lexique (vide au départ). On obtient donc le lexique :

```
[ Mot("la","0/1") ; Mot("vache","1") ; Mot("regarde","(0\S)/2") ;
  Mot("le","2/3") ; Mot("train", "3")]
```

- On réalise ensuite l'unification sur ce lexique : comme il n'y a pas de doublon, il ne change pas.

- On est maintenant à l'étape :

```
unification( creer_lexique( algoRG(hd(listarb),"S",c#incr),"",[])
             @ [ Mot("la","0/1") ; Mot("vache","1") ; Mot("regarde","(0\S)/2") ;
                 Mot("le","2/3") ; Mot("train", "3")]  )
```

On applique l'algorithme RG sur le premier arbre de la liste et on obtient :

```
Noeud("elimg",
 [Arete("4",Noeud("elimd",[Arete("4/5",Noeud("le",[]));
                           Arete("5",Noeud("chat",[]))]));
  Arete("4\S",Noeud("elimd",[ Arete("(4\S)/6",Noeud("regarde",[]));
                              Arete("6",Noeud("elimd",[Arete("6/7",Noeud("la",[]));
                                                       Arete("7",Noeud("vache",[]))
                                                      ]
                                                )
                                       )
                              ]
                    )
       ) ]   )
```

- On applique ensuite la fonction `creer_lexique` sur cet arbre en lui passant également en paramètre une chaîne de caractères vide au départ (dans les appels récursifs, ce sera l'étiquette correspondant au nœud étudié) ainsi qu'une liste correspondant au lexique (vide au départ). On obtient donc le lexique :

```
[ Mot("le","4/5") ; Mot("chat","5") ; Mot("regarde","(4\S)/6") ;
  Mot("la","6/7") ; Mot("vache", "7")]
```

- On réalise ensuite l'unification sur ce lexique car il y a plusieurs doublons :

```
unification(  [ Mot("le","4/5") ; Mot("chat","5") ; Mot("regarde","(4\S)/6") ;
                Mot("la","6/7") ; Mot("vache", "7")]
            @ [ Mot("la","0/1") ; Mot("vache","1") ; Mot("regarde","(0\S)/2") ;
                Mot("le","2/3") ; Mot("train", "3")]
          )
```

On commence par identifier le doublon `Mot("le","4/5")` et `Mot("le","2/3")`.
Remarque : à chaque étape, on identifie les mots deux par deux.

Puis on appelle la fonction `unif` en lui passant en paramètre le lexique ainsi que les types `"4/5"` et `"2/3"`.
La fonction `unif` appelle la fonction `decompose_type` (qui elle-même appelle la fonction `decomp_en_type`) sur `"4/5"` puis sur `"2/3"` et on obtient deux listes composées des décomposition des deux types :

```
[ "4/5" ;  "4"  ;  "5" ]
[ "2/3" ;  "2"  ;  "3" ]
```

Ensuite, la fonction `unif` appelle la fonction `uni` avec le lexique et ces deux listes.
La fonction `uni` effectue toutes les substitutions (fonction `substitution`) en remplaçant le premier type de la première liste par le premier type de la seconde liste, le deuxième type de la première liste par le deuxième type de la seconde liste, etc…
On obtient donc le lexique suivant :

```
[ Mot("le","2/3") ; Mot("chat","3") ; Mot("regarde","(2\S)/6") ;  Mot("la","6/7") ;
  Mot("vache", "7")]  ;  Mot("la","0/1") ;  Mot("vache","1") ;
  Mot("regarde","(0\S)/2") ; Mot("le","2/3") ; Mot("train", "3")  ]
```

Enfin, on supprime les doublons (même mot et même type) et on obtient :

```
[ Mot("chat","3") ; Mot("regarde","(2\S)/6") ; Mot("la","6/7"); Mot("vache", "7")];
  Mot("la","0/1") ; Mot("vache","1") ; Mot("regarde","(0\S)/2") ;
  Mot("le","2/3") ; Mot("train", "3")  ]
```

On réalise toutes ces opérations pour chaque doublon et on obtient le lexique final :

```
[ Mot("la","0/1") ;  Mot("vache","1")  ;  Mot("regarde","(0\S)/0") ;
  Mot("le","0/3") ;  Mot("train", "3") ;  Mot("chat","3");  ]
```

Enfin, on affiche à l'écran et on écrit dans un fichier le lexique sous la forme ci-dessous :

```
la : 0/1
vache : 1
regarde : (0\S)/0
le : 0/3
train : 3
chat : 3
```

5.6 Présentation de l'interface

Une première mouture de cette interface nous a été fournie. Nous l'avons réécrite (en s'inspirant tout de même fortement de l'originale). Une partie de l'interface est à ce jour inexploitée et son implantation semble délicate d'autant que son utilité reste à prouver !

Aussi cette présentation de l'interface s'attachera autant à montrer ce qui a été fait qu'à proposer des améliorations. Nous essaierons de formuler une ébauche de cahier des charges.

L'implémentation de l'interface s'appuie sur l'utilisation de Tcl/Tk. Tk offre la possibilité de construire des fenêtres. Le principe général de fonctionnement de l'interface est de proposer des menus, sous-menus et ainsi de suite jusqu'à l'application de l'algorithme.

5.6.1 Présentation de l'interface actuelle

Menu principal

Le menu principal du programme propose de choisir parmi les différents algorithmes implémentés (à ce jour au nombre de trois : RG, RG* et RLG mais il est facile d'ajouter des boutons pour lancer d'autres algorithmes) :

Capture 1 – Menu principal

Ensuite, l'interface offre deux possibilités pour entrer les données sur lesquelles l'algorithme choisi précédemment sera appliqué, cela peut être fait :

Option 1 - à partir d'un fichier XML existant,

Option 2 - en entrant des exemples à la main, c'est ce second choix qui n'est pas encore implémenté et pose quelques problèmes…

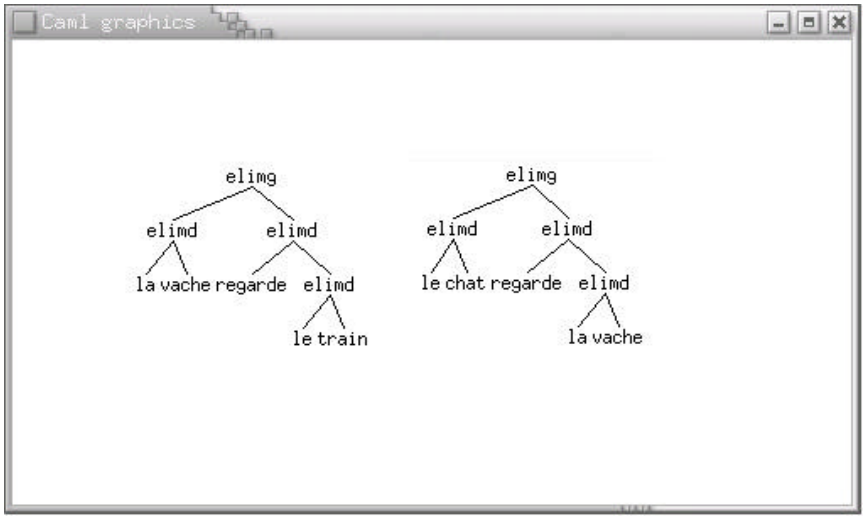
Capture 2 – Choix du mode d'entrée des exemples

Détails de l'option 1

Si on poursuit en choisissant l'***option 1***, une fenêtre s'ouvre avec deux champs texte et trois boutons. Les champs textes accueillent dans l'ordre le nom du fichier XML source et celui du fichier texte cible (dont le format est donné plus haut). Vers la sortie est envoyé le lexique produit par l'application de l'algorithme et de l'unification.

Capture 3 – Appliquer l'algorithme

Les boutons offrent la possibilité d'afficher sous forme graphique les arbres contenus dans le fichier source, de fermer la fenêtre contenant cette représentation graphique et d'appliquer l'algorithme. Enfin, outre son envoi vers le fichier de sortie, le lexique obtenu est également affiché sur la sortie standard.



Capture 4 – Affichage des arbres

Inconvénient de cette partie de l'implémentation de l'interface (option 1) :

La représentation en XML peut accueillir plusieurs arbres. Lorsqu'on applique l'algorithme à un fichier XML, on ne peut le faire qu'à l'ensemble des arbres que contient le fichier. Si on souhaite connaître le lexique à chaque étape (pour chaque nouvel exemple positif), la seule solution est de créer un ensemble de fichiers XML contenant respectivement : le premier exemple, les deux premiers exemples, les trois premiers exemples et ainsi de suite…

Détails de l'option 2

Les fonctionnalités que propose l'***option 2*** ne sont pas encore implémentées. Telle qu'elle est construite, l'interface pour cette option prévoit, dans une première fenêtre, de choisir le nom du fichier où le(s) exemple(s) que l'on rentrera par la suite seront enregistrés.

Capture 5 – Nouveau fichier

Puis, dans une seconde fenêtre, elle prévoit :
- d'enregistrer (c'est à dire d'envoyer vers le fichier donné précédemment) un exemple entré manuellement,
- d'entrer un autre exemple (appel récursif : ouverture d'une fenêtre similaire),
- d'appliquer l'algorithme.

Capture 6 – Nouvel exemple

Inconvénient de cette partie de l'implémentation de l'interface (option 2) :

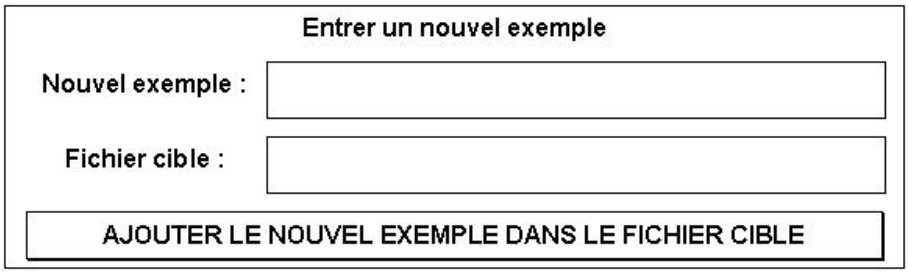
Un exemple rentré dans le champ texte est de type chaîne de caractères : il n'est donc pas directement exploitable par le programme. Cela pose plusieurs questions :
- quel format utilise-t-on en entrée ? On peut écrire les arbres en utilisant le format XML, les types CaML ou encore utiliser une autre représentation… et il faudra écrire la fonction de conversion correspondante.
- dans quel format enregistre-t-on les données dans le fichier cible ? Il semblerait cohérent de les enregistrer au format XML déjà défini. Dans ce cas, il faut écrire une autre fonction de conversion (pour convertir un arbre CaML en arbre XML, ou convertir directement le format d'entrée du champ texte en XML…)

5.6.2 Propositions pour une interface plus adaptée

De l'étude de l'interface implémentée, il se dégage deux fonctionnalités du programme : "possibilité d'entrer manuellement un nouvel exemple" (et prévoir sa persistance) et "appliquer un algorithme" (avec tout ce que cela implique). Dès lors, il semble pragmatique d'orienter la nouvelle interface autour de ces deux fonctionnalités !

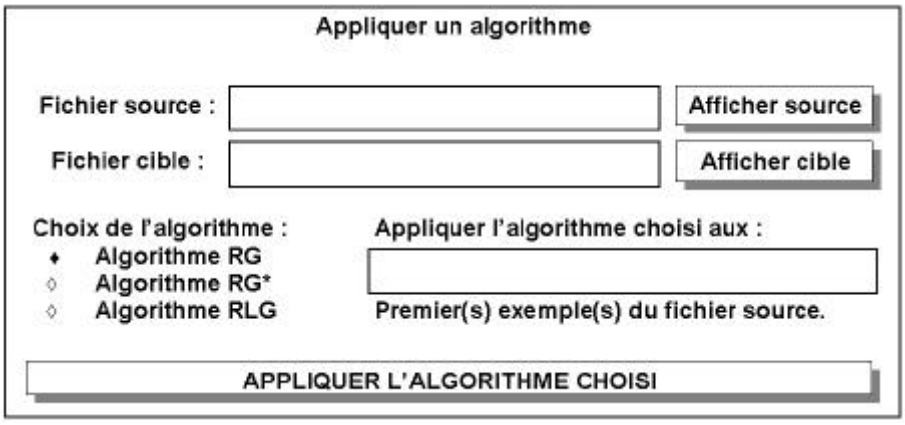
On peut s'interroger sur l'intérêt de proposer à l'utilisateur d'entrer manuellement des exemples dans le programme. Si cette fonctionnalité est nécessaire alors il faut définir un format d'entrée pour l'utilisateur. Il nous semble que le plus simple serait d'utiliser la notation suivante : `[\ Zidane [/ marque [/ un but ]]]`. Il faudra alors prévoir la conversion de ce format vers un arbre XML (ou vers un arbre CaML).

Voici une proposition d'interface pour l'entrée de nouveaux exemples :



Les modifications apportées par cette proposition sont résumables en trois faits :
- l'exemple et le fichier cible figurent sur la même fenêtre (améliore la lisibilité),
- l'ajout d'exemples à un fichier se fait de façon incrémentale,
- l'entrée manuelle de nouveaux exemples est indépendante de l'application de tout algorithme.

Le gros défaut de l'interface actuelle est qu'elle ne permet pas réellement d'appliquer l'algorithme de façon incrémentale. Voici une proposition d'interface qui permettrait de résoudre ce problème :



5.6.3 Conséquences sur la persistance du lexique

On peut tirer divers enseignements de l'étude de l'interface existante et des propositions d'améliorations qui en découlent. Principalement : un format XML a été défini pour la persistance des arbres. Il semblerait cohérent d'utiliser ce format de façon universelle : pour le stockage des entrées comme pour celui des sorties. Mais est-ce vraiment utile ? Un format bien plus simple (et à priori suffisant) peut être proposé pour la persistance du lexique.

On peut se contenter d'un fichier texte comme proposé plus haut. On peut également envisager de créer une autre définition XML, par exemple :

```
<!ELEMENT LEXIQUE (ENTREE)*    >
<!ELEMENT ENTREE  (MOT,TYPE+)  >
<!ELEMENT MOT     (#PCDATA)    >
<!ELEMENT TYPE    (#PCDATA)    >
```

Un lexique est alors constitué d'un ensemble éventuellement vide d'entrées. Une entrée étant elle-même constituée d'un mot et d'au moins un type (on prévoit ici la prise en compte des grammaires *k*-valuées).

L'intérêt de la persistance du lexique est qu'il est alors possible d'en récupérer un déjà créé. De cette façon, il devient possible d'appliquer à nouveau l'algorithme sur un lexique déjà constitué et de nouveaux exemples positifs.

PARTIE 6

Jeux d’essai

Nous présentons ici quelques jeux d’essai montrant le bon fonctionnement des différents algorithmes implémentés. Nous nous attacherons aussi à montrer les limites de tels algorithmes dans le cas où on les applique à des grammaires non rigides.

6.1 Algorithme RG

- On applique l’algorithme RG sur les deux phrases suivantes :

```
[\[/la vache][/regarde [/le train]]] , [\[/le chat][/regarde [/la vache]]]
```

On obtient le lexique suivant :

```
train:3
le:6/3
chat:3
regarde:(6|S)/6
la:6/2
vache:2
```

On vérifie que le lexique est correct en dérivant les phrases :
pour la phrase *la vache regarde le train* :
6/2 , 2 , (6 \ S)/6 , 6/3 , 3 →
6/2 , 2 , (6 \ S)/6 , 6 →
6/2 , 2 , (6 \ S) →
6 , (6 \ S) →
S

On peut aussi engendrer d’autres phrases que les exemples. Ainsi, la phrase *le chat regarde le train* se dérive de la façon suivante :
6/3 , 3 , (6 \ S)/6 , 6/3 , 3 →
6/3 , 3 , (6 \ S)/6 , 6 →
6/3 , 3 , (6 \ S) →
6 , (6 \ S) → S

- On applique l’algorithme RG sur les phrases suivantes :

```
[\ Mary [/ loves John]] , [\ John [/ loves [/ a girl] ]]
     [\ Francesca [/ dances [/ with [/ the boy]]]]
```

On obtient le lexique suivant :

```
Franscesca:7
dances:(7|S)/8
with:8/9
the:9/10
boy:10
a:4/5
girl:5
Mary:4
loves:(4|S)/4
John:4
```

- On applique l’algorithme RG sur les phrases suivantes :

```
[\ [/ a man] swims]  ,  [\ [/ a fish] [\ swims fast] ]
```

On obtient le lexique suivant :

```
a:0/2
fish:2
swims:0|S
fast:(0|S)|(0|S)
man:2
```

6.2 Algorithme RLG

- On applique l’algorithme RLG sur les phrases suivantes :

```
[\ [/I [\ she [/ likes [] ]]]  him ]
   [\ John [/ likes [/ a girl] ]]
```

On obtient le lexique suivant :

```
him:1
John:0
likes:(0|S)/1
a:(1)/2
girl:2
she:(S)/(0|S)
```

- On applique l’algorithme RLG sur les phrases suivantes :

```
[\ Francesca [/ dances [/ with [/ the boy]]]]
        [\ John [/ loves [/ a girl] ]]
           [\ Mary [/ loves John ]]
```

On obtient le lexique suivant :

```
Franscesca:7
dances:(7|S)/8
with:8/9
the:9/10
boy:10
a:4/5
girl:5
Mary:4
loves:(4|S)/4
John:4
```

- On applique l'algorithme RLG sur les phrases suivantes :

```
[\ [/ [/le chat] [/ que [/I [\ je [/ regarde [] ]]]]] dort]
          [\ [/ le chat] [/ regarde [/ la vache]]]
```

On obtient le lexique suivant :

```
dort:0|(S)
le:(0)/6
chat:6
regarde:(0|S)/1
la:(1)/2
vache:2
que:(6|6)/(S/1)
je:0
```

6.3 Algorithme RG*

- On applique l'algorithme RG* sur les phrases suivantes :

```
          [\ [\ le [\*ε [\*it joli [\*it petit chien ] ]]] dort]
[\[\le [\*ε[\*it joli [\*it petit chien ]]]][/*ε[/*it dort silencieusement]]]
```

On obtient le lexique suivant :

```
le:1
joli:2
petit:2
chien:2|*(1|0)
dort:(0|S)/*1
silencieusement:1
```

- On applique l'algorithme RG* sur les phrases suivantes :

```
          [\ [\ le [\*ε [\*it joli [\*it petit chien ] ]]] dort]
[\[\le [\*ε[\*it joli [\*it petit chien ]]]][/*ε[/*it dort silencieusement]]]
     [\ Pierre [/ [/*ε [/*it regarde silencieusement]] [\ le chat] ]]
```

On obtient le lexique suivant :

```
chat:1|7
le:1
joli:2
petit:2
chien:2|*(1|0)
dort:(0|S)/*1
silencieusement:1
Pierre:6
regarde:((6|S)/7)/*1
```

6.4 Contre-exemples

- On applique l'algorithme RG sur les phrases suivantes :

```
[\ il [/ montre [/ le train]]]
[\ [/ ma montre] [/ est noire]]
[\ il [/ montre [/ ma montre]]]
```

On obtient le lexique suivant :

```
le:9/10
train:10
est:(9|S)/5
noire:5
ma:9/((((0|S))/9))
montre:((0|S))/9
il:0
```

Ici, on a appliqué l'algorithme sur des phrases contenant des homonymes qui n'ont pas le même type : *montre* est un verbe dans la première et la troisième phrase alors qu'il est un nom dans la deuxième te la troisième phrase. La grammaire cible ne devrait donc pas être rigide : or ici, elle l'est puisqu'on a effectué l'unification sur les différents types de *montre*. Ceci pose un réel problème puisque la grammaire obtenue peut engendrée des phrases incorrectes telles que * *le train est noire.*

- On applique l'algorithme RLG sur les phrases suivantes :

```
[\ [/ [/le chat] [/ que [/I [\ je [/ regarde [] ]]]]] dort]
            [/ [\ il croit] [/ que [\ je dors]]]
```

On obtient le lexique suivant :

```
regarde:(2|13)/15
dort:9|(S)
il:6
croit:6|(S/(11|9))
que:(11|9)/(13/15)
je:2
dors:2|((13/15))
le:(11)/20
chat:20
```

Ici, on a appliqué l'algorithme sur des phrases contenant des homonymes qui n'ont pas le même type : *que* est un pronom relatif dans la première phrase alors qu'il est une conjonction de subordination dans la seconde. La grammaire cible ne devrait donc pas être rigide : or ici, elle l'est puisqu'on a effectué l'unification sur les différents types de *que*. La grammaire obtenue peut alors engendrée des phrases incorrectes.

6.5 Remarques

Les trois algorithmes d'étiquetage sont complètement implémentés et ont été correctement testés. En revanche, l'unification n'a pas été testée sur suffisamment d'exemples pour être totalement garantie. On se préoccupera en particulier de la gestion des parenthèses qui peut certainement être améliorée.

Conclusion

Nous avons vu que les grammaires catégorielles avec ou sans itérations et les grammaires de Lambek, entièrement lexicalisées, avaient de réelles possibilités d'apprentissage. Les grammaires catégorielles classiques et de Lambek ont en outre un lien étroit avec les systèmes logiques ce qui facilite l'étude de leurs propriétés mathématiques. Les grammaires catégorielles avec itérations, quant à elles, doivent pouvoir être intégrées dans cet aspect pour être vraiment intéressantes. Ces dernières ont aussi quelques limites car si elles permettent, entre autres, de gérer l'accumulation des adjectifs, elles ne peuvent effectuer aucun contrôle sur l'ordre de ceux-ci.

L'approche retenue pour l'apprentissage de ces grammaires est de supposer que les exemples sont fournis avec une structure. Cette hypothèse est linguistiquement réaliste, mais sa mise en œuvre nécessite de structurer tous les exemples en indiquant à chaque fois les règles qui doivent être utilisées pour les réductions et ceci peut devenir très complexe notamment pour les grammaires avec itérations.

Or le but du modèle de Gold est de rapprocher l'apprentissage informatique de l'acquisition du langage par un enfant et il paraît improbable qu'un enfant reçoive des informations aussi structurées, même en tenant compte de la prosodie ou de la ponctuation. On pourrait donc, comme Kanazawa, essayer de diminuer l'importance des informations contenues dans les exemples en appliquant les algorithmes d'apprentissage sur des structures d'arbre non étiqueté ou directement sur des phrases dont la structure ne serait pas fournie. Ceci obligerait à considérer toutes les règles possibles à une étape donnée dans le premier cas ou toutes les structures possibles pour une phrase dans le second cas. Mais on imagine bien l'augmentation considérable de la complexité de l'algorithme que cela entraînerait …

Le même problème se pose si on veut appliquer ces algorithmes aux grammaires catégorielles *k*-valuées : il faudrait essayer toutes les possibilités d'unification des types pour avoir au plus *k* types par mot.

Les perspectives d'amélioration de notre implémentation serait d'envisager l'unification modulo associativité, d'implémenter d'autres algorithmes d'apprentissage et de retravailler l'interface.

En conclusion, l’apprentissage est aujourd’hui au cœur de la problématique des sciences cognitives et plus généralement de l’Intelligence Artificielle. Or les formalismes grammaticaux classiques fournissent peu de modèles d’apprentissage. La réponse à ces difficultés pourrait se trouver dans l’étude des grammaires catégorielles dont on connaît les qualités d’apprentissage. En outre, elles offrent de nombreuses perspectives d’application tant dans l’étude du génome que dans le diagnostic ou le traitement automatique des langues. En particulier, elles semblent bien adaptées à la génération et l’analyse de phrases, utiles par exemple pour les logiciels d’aide à la traduction.

Bibliographie

[1] R. BONATO, *A study of learnability of rigid Lambek grammars*. Tesi di Laurea, Universita di Verona, Université Rennes 1, 2000.

[2] W. BUSZKOWSKI, G. PENN, *Categorial grammars determined from linguistic data by unification*, 1990.

[3] E. M. GOLD, *Language identification in the limit*. Information and control, 10 :447-474, 1967.

[4] M. KANAZAWA, *Learnable classes of categorial grammars*. Studies in Logic, Language and Information. FOLLI, CSLI, Stanford, California, 1998.

[5] I. MEL'CUK, *Dependency in Linguistic Description.*

[6] E. MOREAU, *Apprentissage des grammaires catégorielles et de dépendances*, Mémoire de DEA, 2001

[7] J. NICOLAS, *Grammatical inference as unification*. Technical report 3632, IRISA, Unité de recherche, INRIA Rennes, March 1999.

[8] C. RÉTORÉ, *The logic of categorial grammars*. Lecture notes ESSLLI'2000, Birmingham, 2000.

[9] C. RÉTORÉ, *Systèmes déductifs et traitement des langues : un panorama des grammaires catégorielles*. Rapport de recherche INRIA n° 3917, avril 2000.

ANNEXE A

Manuel d'utilisation

1. OCaML 3.04 doit être installé avec les librairies Findlib, Netstring et Pxp.
   Lancer l'interpréteur avec la ligne de commande :

```
ocaml -I +labltk -I +site-lib/pxp-engine -I +site-lib/pxp-lex-iso88591 -I
+site-lib/pxp-lex-utf8 -I +site-lib/netstring
```

2. Taper la ligne :

```
#use "header" ;;
```

pour ouvrir et charger les modules nécessaires ainsi que les fichiers :
- `sortie.ml`
- `algorithmes.ml`
- `interface.ml`

3. Taper `programme();;` pour lancer l'interface.

4. Cliquer sur le bouton correspondant à l'algorithme souhaité.

5. Cliquer sur le bouton « `Charger un fichier déjà existant` ».

6. Renseigner le premier champ par le nom d'un fichier XML existant et contenant les exemples. Vous pouvez déjà afficher les arbres ainsi chargés en cliquant sur « `Afficher les arbres` ». Fermer *obligatoirement* l'affichage en cliquant sur « `Fermer l'affichage` ».

7. Renseigner le second champ avec le nom du fichier de sortie. Si ce fichier existe déjà, il sera écrasé.

8. Vous pouvez maintenant appliquer l'algorithme en cliquant sur « `Appliquer l'algorithme` ».

9. Vous pouvez appliquer à nouveau le même algorithme en entrant un nouveau fichier d'exemples et ceci sans fermer les fenêtres.

Remarque :
l'application effective des algorithmes ne se fait que quand toutes les fenêtres ont été fermées.

ANNEXE B

Listing commenté

```
open Tk;;
open Graphics;;
open Pxp_yacc;;
open Pxp_document;;
open List;;
open String;;
open Str;;

#load "str.cma";;
#load "graphics.cma";;
#load "unix.cma";;
#load "labltk.cma";;
#load "libjpf.cma";;
#load "/usr/local/lib/ocaml/site-lib/netstring/netstring.cma";;
#load "pxp_engine.cma";;
#load "pxp_lex_utf8.cma";;
#load "pxp_lex_iso88591.cma";;
#load "pxp_lex_link_iso88591.cmo";;

#use "sortie.ml";;
#use "algorithmes.ml";;
#use "interface.ml";;
```

```
(*-----------------------------------------------------------------------------
      Creation de la classe node_extension pour les noeuds XML
-----------------------------------------------------------------------------*)

class node_extension =
  object (self)

    val mutable node = (None : node_extension node option)
    method clone = {< >}
    method node = match node with None -> assert false | Some n -> n
    method set_node n = node <- Some n

end ;;


(*-----------------------------------------------------------------------------
       Specifier que node_extension est l'extension des noeuds XML
-----------------------------------------------------------------------------*)
let spec =
  make_spec_from_alist
   ~data_exemplar: ( new data_impl (new node_extension))
   ~default_element_exemplar: (new element_impl (new node_extension))
   ~element_alist: []();;

(*-----------------------------------------------------------------------------
       Definir un environnement de trace pour l'arbre
-----------------------------------------------------------------------------*)
class environnement_graphique =
  object (self)

    method get_sep_vert_entre_dag = 10   (* Espace Vert. entre les dags *)
    method get_sep_entre_pere_fils = 15  (* Espace Vert. entre un pere et ses fils *)
    method get_sep_entre_fils = 3        (* Espace entre les boites des deux fils (deux freres) *)
    method largeur_chaine str = fst (self#taille_chaine str)
    method hauteur_chaine str = snd (self#taille_chaine str)
    method taille_chaine str = Graphics.text_size str
    method debut_trace = open_graph ""
    method fin_trace = ()
    method tracer_ligne x0 y0 x1 y1 =
      Printf.printf "tracer_ligne %d %d %d %d\n"  x0 y0 x1 y1;
```

```
      Graphics.moveto x0 y0; Graphics.lineto x1 y1
    method tracer_string x y s =
      Printf.printf "tracer_string %d %d >%s< \n" x y s;
      let (l,h) = Graphics.text_size s in Graphics.moveto (x - l/2) (y - h/2);
                                          Graphics.draw_string s
end ;;

(*-------------------------------------------------------------------------
    Fonctions de gestion d'erreur si la structure XML n'est pas correcte
-------------------------------------------------------------------------*)

let erreur_balise_incorrecte f s e = match e#node_type with
    Pxp_document.T_element b -> failwith (f ^ ": balise <" ^ b ^ "> trouvee au lieu de <" ^ s ^ ">")
  | _ -> failwith (f ^ ": autre element trouve au lieu de la balise <" ^ s ^ ">") ;;

let erreur_liste_balises_incorrecte f s e =
    failwith (f ^ ": liste de noeud(s)de longueur " ^ string_of_int(List.length e) ^
                                          " trouvee au lieu de <" ^ s ^ ">") ;;

let erreur_noeud_incorrect f s e =
    failwith (f ^ ": type de noeud trouve incorrect au lieu de <" ^ s ^ ">") ;;

(*-------------------------------------------------------------------------
    Transformation d'un document XML e arbre
         XML: <DOCUMENT> = <DAG>+ ...
              <DAG> = <NOEUD>
              <NOUEUD> = <ETIQ> ( <ETIQ>?, <NOEUD>)*
              <ETIQ> = <#PCDATA>
        Arbre: type 'a arbre = Noeud of ('a, 'a arete list)
               and  'a arete = Arete of ('a, 'a arbre)
-------------------------------------------------------------------------*)
type 'a arbre = Noeud of ('a * 'a arete list)
and  'a arete = Arete of ('a * 'a arbre) ;;

(*-------------------------------------------------------------------------
   Descente recursive de l'arbre XML dag pour creer un string arbre list
-------------------------------------------------------------------------*)
let creer_arbre (document:'a document) =
(* ------ Balise <document> = dag+ -------- *)
let rec t_document n = t_document_0 (n#sub_nodes)
and t_document_0 l = match l with
```

```
     []  -> []
  | h::t -> t_document_1 h @  t_document_0 t
and t_document_1 h = match (h#node_type) with
    Pxp_document.T_element "DAG" -> [t_dag h]
  |             _                -> []

(* ------ Balise <dag> = noeud -------- *)
and t_dag n = t_dag_0 (n#sub_nodes)
and t_dag_0 l = match l with
    h::t -> t_dag_1 h t
  |  []  -> erreur_liste_balises_incorrecte "creer_arbre:t_dag_0" "<noeud>" l
and t_dag_1 n t = match n#node_type with
    Pxp_document.T_element "NOEUD" -> t_noeud n
  |             _                  ->  t_dag_0 t
(* ------ Balise <noeud> = etiq,(etiq?,noeud)* -------- *)
and t_noeud n = t_noeud_0 (n#sub_nodes)
and t_noeud_0 l = match l with
     []  -> erreur_liste_balises_incorrecte "creer_arbre:t_noeud_0" "<etiq>,(etiq?,noeud)*" l
  | h::t -> t_noeud_1 h t
and t_noeud_1 n t = match n#node_type with
    Pxp_document.T_element "ETIQ" -> Noeud(t_etiq n, t_noeud_2 t)
  |             _                 -> t_noeud_0 t
and t_noeud_2 l = match l with
     []  -> []
  | h::t ->  t_noeud_3 h t
and t_noeud_3 n t = match n#node_type with
    Pxp_document.T_element "ETIQ"  -> t_noeud_4 (t_etiq n) t
  | Pxp_document.T_element "NOEUD" -> (Arete("",  t_noeud n)) :: (t_noeud_2 t)
  |             _                  -> t_noeud_2 t
and t_noeud_4 etiq l = match l with
     []  -> erreur_liste_balises_incorrecte "creer_arbre:t_noeud_4" "etiq,(etiq?,<noeud>)*" l
  | h::t -> t_noeud_5 etiq h t
and t_noeud_5 etiq n t = match n#node_type with
  | Pxp_document.T_element "NOEUD" -> (Arete(etiq,  t_noeud n)) :: (t_noeud_2 t)
  |             _                  -> t_noeud_4 etiq t

(* ------ Balise <etiq> = #PCDATA -------- *)
and t_etiq n = t_etiq_0 (n#sub_nodes)
and t_etiq_0 l = match l with
    h::t -> t_etiq_1 h t
  |  []  -> ""
```

```
and t_etiq_1 n t = match n#node_type with
    Pxp_document.T_data -> (n#data) ^ (t_etiq_0 t)
  |              _       -> t_etiq_0 t
(* ------ Appel a la fonction recursive ----- *)
in   t_document document#root
;;

(*----------------------------------------------------------------------------
       Definir un environnement de trace pour l'arbre
----------------------------------------------------------------------------*)
class environnement_graphique =
  object (self)

    method get_sep_vert_entre_dag = 10   (* Espace Vert. entre les dags *)
    method get_sep_entre_pere_fils = 20  (* Espace Vert. entre un pere et ses fils *)
    method get_sep_entre_fils = 3        (* Espace entre les boites des deux fils (deux freres) *)
    method largeur_chaine str = fst (self#taille_chaine str)
    method hauteur_chaine str = snd (self#taille_chaine str)
    method taille_chaine str = Graphics.text_size str
    method debut_trace = open_graph ""
    method fin_trace = ()
    method tracer_ligne x0 y0 x1 y1 =
      Printf.printf "tracer_ligne %d %d %d %d\n"  x0 y0 x1 y1;
      Graphics.moveto x0 y0; Graphics.lineto x1 y1
    method tracer_string x y s =
      Printf.printf "tracer_string %d %d >%s< \n" x y s;
      let (l,h) = Graphics.text_size s in Graphics.moveto (x - l/2) (y - h/2);
                                          Graphics.draw_string s
end ;;


(*----------------------------------------------------------------------------
    Fonction auxiliaire pour ajouter de l'espace a une taille non-nulle
----------------------------------------------------------------------------*)
let ajout_espace taille espace = if taille == 0 then 0 else taille+espace ;;
let ajout_espace2 taille1 taille2 espace =
  if taille1 == 0 then taille2
  else if taille2 == 0 then taille1 else taille1+taille2+espace ;;


(*----------------------------------------------------------------------------
```

```
   Descente recursive de l'arbre pour trouver la taille des boites/sous-arbres1
-------------------------------------------------------------------------------*)
let calcul_taille_arbre (g:environnement_graphique) dags =
(* ------ liste de dags -------- *)
let rec t_dags dags = match dags with
     []  -> (0,0,[])
  | h::t -> let (xh,yh,ah as a) = t_noeud h in
            let (xt,yt,at) = t_dags t in
               (max xh xt,
                ajout_espace2 yh yt (g#get_sep_vert_entre_dag),
                ah::at)
(* ------ un noeud -------- *)
and t_noeud (Noeud(etiq, aretes)) =
    let (xe,ye) = g#taille_chaine etiq in
    let (xa,ya,la) = t_liste_aretes aretes in
    let x = max xe xa in
    let y = ye + ya in
                (x, y, Noeud((xe,ye,xa,ya,etiq),la))
(* ------ une listes d'aretes ------ *)
and t_liste_aretes aretes = match aretes with
     []  -> (0, 0, [])
  | h::t -> let (xh,yh,ah) = t_arete h in
            let (xt,yt,at) = t_liste_aretes t in
              (ajout_espace2 xh xt (g#get_sep_entre_fils), max yh yt, ah::at)
(* ------ une arete -------- *)
and t_arete (Arete(etiq, noeud)) =
    let (xe,ye) = g#taille_chaine etiq in
    let (xn,yn,n) = t_noeud noeud in
    let y = ajout_espace yn (g#get_sep_entre_pere_fils) in
      (xn, y , Arete((xe,ye,xn,yn,etiq),n))
(* ------ appel a la fonction recursive ----- *)
in   t_dags dags
;;

(*-------------------------------------------------------------------------
   Descente recursive de l'arbre pour le tracer (avec tailles incluses)
-------------------------------------------------------------------------*)
let tracer_arbre (g:environnement_graphique) x y dags =
(* ------ liste de dags -------- *)
let rec t_dags x y dags = match dags with
     []  -> ()
```

```
  | a::t -> let h = t_noeud x y a in
                  t_dags x (y - ajout_espace h (g#get_sep_vert_entre_dag)) t
(* ------ un noeud -------- *)
and t_noeud x y (Noeud((we,he,wa,ha,etiq), aretes)) =
    let x_etiq = x in
    let y_etiq = y - he/2 in
    let x_point_etiq = x in
    let y_point_etiq = y - he in
    let xg_fils = x - wa/2 in
    let y_fils = y_point_etiq in
      g#tracer_string x_etiq y_etiq etiq;
      t_liste_aretes x_point_etiq y_point_etiq xg_fils y_fils aretes;
      he+ha

(* ------ une listes d'aretes ------ *)
and t_liste_aretes x_point_etiq y_point_etiq xg_fils y_fils aretes = match aretes with
     []  -> ()
  | h::t -> let w = t_arete x_point_etiq y_point_etiq xg_fils y_fils h in
              t_liste_aretes x_point_etiq y_point_etiq
                             (xg_fils + ajout_espace w (g#get_sep_entre_fils)) y_fils t
(* ------ une arete -------- *)
and t_arete x_point_pere y_point_pere xg_fils y_fils (Arete((we,he,wn,hn,etiq), noeud)) =
    let x_noeud = xg_fils + wn/2 in
    let y_noeud = y_fils - (g#get_sep_entre_pere_fils) in
    let x_etiq = (x_noeud + x_point_pere)/2 in
    let y_etiq = (y_noeud + y_point_pere)/2 in
    let hn = t_noeud x_noeud y_noeud noeud in
      g#tracer_ligne x_point_pere y_point_pere x_noeud y_noeud;
      g#tracer_string x_etiq y_etiq etiq;
      wn
(* ------ appel a la fonction recursive ----- *)
in  let (w,h,l) = calcul_taille_arbre g dags in
      t_dags (x+w/2) (y+h) l
;;
```

```
(* ************************************************************** *)
(* ** utilitaire ********************************************** *)
(* **** compteur : variable globale pour compter les étiquettes ***** *)
(* ************************************************************** *)

class compteur = object
  val mutable cpt = 0            (* initialisation du compteur à 0         *)
  method get = cpt               (* renvoie la valeur courante du compteur *)
  method reset = cpt <- 0        (* remise à 0 du compteur                 *)
  method incr = cpt <- cpt+1     (* incrémentation du compteur             *)
  method nothing = cpt <- cpt    (* ne fait rien !                         *)
end  ;;

(* *** compteur *** *)
let c =  new compteur;;


(* ************************************************************** *)
(* ************************************************************** *)
(* ** TYPES *************************************************** *)
(* ************************************************************** *)
(* ************************************************************** *)
(* ** type arbre : noeud et une liste d'aretes ********************** *)

(*
type 'a arbre = Noeud of ('a * 'a arete list)
and  'a arete = Arete of ('a * 'a arbre) ;;
*)

(* ** type "lexique" pour stocker tous les mots ******************** *)
(* **** feuilles de l'arbre et leur(s) type(s) ********************** *)

type 'a lexeme = Mot of ('a * 'a );;
type 'a lexique = 'a lexeme list;;


(* ************************************************************** *)
(* ************************************************************** *)
(* ************************************************************** *)
(* ** GRAMMAIRE AB ******************************************** *)
(* ************************************************************** *)
```

```
(* ****************************************************************** *)
(* ****************************************************************** *)

(* ****************************************************************** *)
(* ** Algorithme RG (pour un seul arbre) **************************** *)
(* **** prend en entrée un arbre, une étiquette et un compteur ****** *)
(* **** renvoie l'arbre étiqueté ************************************ *)
(* ****************************************************************** *)

let rec algoRG = function Noeud(q,l),s , num ->
 if l = []
then (* cas d'arrêt : liste d'aretes vide = c'est une feuille : on renvoie le noeud tel quel *)
     Noeud(q,l)
else let [Arete(x,Noeud(y,z));Arete(t,Noeud(m,p))] = l in
     match q with
      | "elimd" ->
          let numero = c#get in
          (* affectation de l'étiquette aux aretes et rappel de l'algo sur les noeuds fils *)
          (* incrémentation du compteur pour numéroter les noeuds *)
          Noeud(q,[Arete(s^"/"^string_of_int(numero), algoRG(Noeud(y,z),"("^s^"/"^string_of_int(numero)
^")",c#nothing));
                   Arete(string_of_int(numero), algoRG(Noeud(m,p),string_of_int(numero),c#incr))])

     | "elimg" ->
          let numero = c#get in
          Noeud(q,[Arete(string_of_int(numero), algoRG(Noeud(y,z),string_of_int(numero),c#nothing));
                   Arete(string_of_int(numero)^"|"^s, algoRG(Noeud(m,p),"("^string_of_int(numero)
^"|"^s^")",c#incr))])
;;


(* ****************************************************************** *)
(* ****************************************************************** *)
(* ****************************************************************** *)
(* ** GRAMMAIRE DE LAMBEK ******************************************* *)
(* ****************************************************************** *)
(* ****************************************************************** *)
(* ****************************************************************** *)
(* ****************************************************************** *)

(* ****************************************************************** *)
(* ** utilitaire **************************************************** *)
```

```
(* ** renvoie le type d'une arête ********************************** *)
(* ************************************************************** *)

let type_arete = function[Arete(x,Noeud(y,z))] -> x;;


(* ************************************************************** *)
(* ** sert pour l'étape de normalisation ************************** *)
(* **** remplace la feuille déchargée par un arbre ***************** *)
(* **** prend en entrée deux arbres et un type d'intro ************** *)
(* ************************************************************** *)

let rec remplace_arbre = function Noeud(i,j), intro, a ->
(* une liste d'aretes contient au moins une arete et au plus deux *)
if tl(j) = []
then (* il n'y a qu'une arete dans la liste *)
     let [Arete(x,Noeud(y,z))] = j in
     if y = "[]"
     then Noeud(i,[Arete(x,a)])
     else remplace_arbre(Noeud(y,z),intro,a)
else (* il y a deux aretes dans la liste *)
     let [Arete(x,Noeud(y,z)) ; Arete(t,Noeud(m,p))] = j in
     if y = "[]" && intro = "introg"
     then Noeud(i,[Arete(x,a) ; Arete(t,Noeud(m,p))] )
     else if m = "[]" && intro = "introd"
          then Noeud(i,[Arete(x,Noeud(y,z)) ; Arete(t,a)] )
          else if intro = "introg"
               then Noeud(i,[Arete(x,remplace_arbre(Noeud(y,z),intro,a)) ; Arete(t,Noeud(m,p))])
               else Noeud(i,[Arete(x,Noeud(y,z)) ; Arete(t,remplace_arbre(Noeud(m,p),intro,a))])
;;


(* ************************************************************** *)
(* ** Normalisation d'un arbre ************************************ *)
(* ************************************************************** *)

let rec normalisation = function Noeud(q,l) ->
if l = []
then Noeud(q,l)
else if q = "introg"
     then let Noeud(q,[Arete(x,Noeud(y,z))]) = Noeud(q,l) in
```

```
          let [Arete(a,Noeud(b,c));Arete(d,Noeud(e,f))] = z in
          if y = "elimg" &&  b = "[]" && c = []
          then normalisation(Noeud(e,f))
          else Noeud(q,[Arete(x,normalisation(Noeud(y,z)))])
     else if q = "introd"
          then let Noeud(q,[Arete(x,Noeud(y,z))]) = Noeud(q,l) in
               let [Arete(a,Noeud(b,c));Arete(d,Noeud(e,f))] = z in
               if y = "elimd" &&  e = "[]" && f = []
               then normalisation(Noeud(b,c))
               else Noeud(q,[Arete(x,normalisation(Noeud(y,z)))])
          else if q = "elimg"
               then let Noeud(q,[Arete(x,Noeud(y,z));Arete(t,Noeud(m,p))]) = Noeud(q,l) in
                    if m = "introg"
                    then let [Arete(s,Noeud(i,j))] = p in
                         normalisation(remplace_arbre(Noeud(i,j),"introg",Noeud(y,z)))
                    else Noeud(q,[Arete(x,normalisation(Noeud(y,z)));Arete(t,normalisation(Noeud(m,p)))])
               else let Noeud(q,[Arete(x,Noeud(y,z));Arete(t,Noeud(m,p))]) = Noeud(q,l) in
                    if y = "introd"
                    then let [Arete(s,Noeud(i,j))] = z in
                         normalisation(remplace_arbre(Noeud(i,j),"introd",Noeud(m,p)))
                    else Noeud(q,[Arete(x,normalisation(Noeud(y,z)));Arete(t,normalisation(Noeud(m,p)))])
;;


(* ************************************************************** *)
(* ** Typage des noeuds arguments d'un arbre ********************** *)
(* ************************************************************** *)

let rec argument = function Noeud(q,l), num ->
if l = []   (*cas d'arrêt : liste d'aretes vide ie c'est une feuille : on renvoie le noeud tel quel *)
then Noeud(q,l)
else if q="elimd"
     then let [Arete(x,Noeud(y,z));Arete(t,Noeud(m,p))] = l in
          let numero = c#get in
          if m = "introg" || m = "introd"     (* on regarde le noeud à droite *)
          then Noeud(q,[Arete(x, argument(Noeud(y,z),c#incr)); Arete(t, argument(Noeud(m,p),c#incr))])     (*
pas de typage *)
          else (* il faut typer l'arete de droite mais pas celle de gauche *)
                Noeud(q,[Arete(x, argument(Noeud(y,z),c#incr)); Arete(string_of_int(numero), argument(Noeud
(m,p),c#incr))])
     else if q = "elimg"
```

```
          then let [Arete(x,Noeud(y,z));Arete(t,Noeud(m,p))] = l in
               let numero = c#get in
               if y = "introg" || y = "introd"     (* on regarde le noeud à gauche *)
               then Noeud(q,[Arete(x, argument(Noeud(y,z),c#incr)); Arete(t,argument(Noeud(m,p),c#incr))])
(* pas de typage *)
               else (* il faut typer l'arete de gauche mais mais pas celle de droite *)
                    Noeud(q,[Arete(string_of_int(numero),argument(Noeud(y,z),c#incr)); Arete(t,argument(Noeud
(m,p),c#incr))])
          else (* on doit typer l'arete *)
               if q = "introd"
               then let [Arete(x,Noeud(y,z))] = l in
                    let numero = c#get in
                    Noeud(q,[Arete(string_of_int(numero), argument(Noeud(y,z),c#incr))])
               else let [Arete(x,Noeud(y,z))] = l in
                    let numero = c#get in
                    Noeud(q,[Arete(string_of_int(numero), argument(Noeud(y,z),c#incr))])
;;


(* ************************************************************** *)
(* ** sert pour l'Algorithme RLG ********************************** *)
(* **** prend en entrée une liste d'aretes et un type d'intro ******* *)
(* **** renvoie le type introduit par introg ou introd ************** *)
(* ************************************************************** *)

let rec type_intro = function l, intro ->
if tl(l) = []
then (* il n'y a qu'une arete dans la liste *)
     let [Arete(x,Noeud(y,z))] = l in
     if y = "[]"
     then x
     else type_intro(z,intro)
else let [Arete(x,Noeud(y,z)) ; Arete(t,Noeud(m,p))] = l in
     if y = "[]" && intro = "introg"
     then x
     else if m = "[]" && intro = "introd"
          then t
          else if intro = "introg"
               then type_intro(z,intro)
               else type_intro(p,intro)
;;
```

```
(* ************************************************************** *)
(* ** Algorithme RLG pour un seul arbre ************************** *)
(* **** prend en entrée un arbre, une étiquette et un compteur ****** *)
(* **** renvoie l'arbre étiqueté ********************************** *)
(* ************************************************************** *)

let rec algoRLG = function Noeud(q,l),s ->
if l = []
then (* cas d'arrêt : c'est une feuille *)
     Noeud(q,l)
else if q="elimd"
     then let [Arete(x,Noeud(y,z));Arete(t,Noeud(m,p))] = l in
          if m = "introg"
          then (* le noeud suivant est de type "introg" : il faut récupérer le type du mot "[]" *)
               let type_de_arete = type_arete(p)^"|"^type_intro(p,"introg") in
               Noeud(q,[Arete("("^s^")/("^type_de_arete^")", algoRLG(Noeud(y,z),s^"/("^type_de_arete^")"));
                        Arete(type_de_arete,algoRLG(Noeud(m,p),"("^type_de_arete^")"))])
          else if m = "introd"   (* le noeud suivant est de type "introd" : il faut récupérer le type du mot
"[]" *)
               then let type_de_arete = type_arete(p)^"/"^type_intro(p,"introd") in
                    Noeud(q,[Arete("("^s^")/("^type_de_arete^")", algoRLG(Noeud(y,z),s^"/
("^type_de_arete^")"));
                             Arete(type_de_arete,algoRLG(Noeud(m,p),"("^type_de_arete^")"))])
             else (* typage normal *)
                    Noeud(q,[Arete("("^s^")/"^t, algoRLG(Noeud(y,z),s^"/"^t)); Arete(t, algoRLG(Noeud
(m,p),t))])
     else if q = "elimg"
          then let [Arete(x,Noeud(y,z));Arete(t,Noeud(m,p))] = l in     (* la liste des aretes est de ce type
*)
               if y = "introg"
               then let type_de_arete = type_arete(p)^"|"^type_intro(z,"introg") in
                    Noeud(q,[Arete("("^type_de_arete^")", algoRLG(Noeud(y,z),type_de_arete));
                             Arete("("^type_de_arete^")|("^s^")",algoRLG(Noeud(m,p),"("^type_de_arete^")
|"^s))])
             else if y = "introd"
                    then let type_de_arete = type_arete(z)^"/"^type_intro(z,"introd") in
                         Noeud(q,[Arete("("^type_de_arete^")", algoRLG(Noeud(y,z),type_de_arete));
                                  Arete("("^type_de_arete^")|("^s^")",algoRLG(Noeud(m,p),"("^type_de_arete^")
|"^s))])
```

```
                  else Noeud(q,[Arete(x, algoRLG(Noeud(y,z),x));
                                Arete(x^"|("^s^")", algoRLG(Noeud(m,p),x^"|"^s)) ])
          else (* c'est un noeud introductif : on passe directement au typage du fils *)
                let [Arete(x,Noeud(y,z))] = l in
                Noeud(q,[Arete(x,algoRLG(Noeud(y,z),x))])
;;


(* ************************************************************** *)
(* ************************************************************** *)
(* ************************************************************** *)
(* ** GRAMMAIRES CATEGORIELLES AVEC ITERATION ******************** *)
(* ************************************************************** *)
(* ************************************************************** *)
(* ************************************************************** *)

(* ************************************************************** *)
(* ** Algorithme RG* pour un seul arbre ************************** *)
(* **** prend en entrée un arbre, une étiquette et un compteur ****** *)
(* **** renvoie l'arbre étiqueté ********************************** *)
(* ************************************************************** *)

let rec algoRGetoile = function Noeud(q,l),s , num ->
if l = []
then (* cas d'arrêt : c'est une feuille *)
     Noeud(q,l)else match q with
     | "elimd" ->
          let [Arete(x,Noeud(y,z));Arete(t,Noeud(m,p))] = l in
          let numero = c#get in
          Noeud(q,[Arete(s^"/"^string_of_int(numero), algoRGetoile(Noeud(y,z),"("^s^"/"^string_of_int(numero)
^")",c#nothing));
                   Arete(string_of_int(numero), algoRGetoile(Noeud(m,p),string_of_int(numero),c#incr))])
     | "elimg" ->
          let [Arete(x,Noeud(y,z));Arete(t,Noeud(m,p))] = l in
          let numero = c#get in
          Noeud(q,[Arete(string_of_int(numero), algoRGetoile(Noeud(y,z),string_of_int(numero),c#nothing));
                   Arete(string_of_int(numero)^"|"^s, algoRGetoile(Noeud(m,p),"("^string_of_int(numero)
^"|"^s^")",c#incr))])
     | "itd" ->
          let [Arete(x,Noeud(y,z));Arete(t,Noeud(m,p))] = l in
          let numero = c#get in
```

```
          Noeud(q,[Arete(s, algoRGetoile(Noeud(y,z),s,c#nothing));
                   Arete(string_of_int(numero), algoRGetoile(Noeud(m,p),string_of_int(numero),c#nothing))])
     | "itg" ->
          let [Arete(x,Noeud(y,z));Arete(t,Noeud(m,p))] = l in
          let numero = c#get in
          Noeud(q,[Arete(string_of_int(numero), algoRGetoile(Noeud(y,z),string_of_int(numero),c#nothing));
                   Arete(s, algoRGetoile(Noeud(m,p),s,c#nothing))])
     | "epsd" ->
          let [Arete(x,Noeud(y,z))] = l in
          let numero = c#get in
          Noeud(q,[Arete(s^"/*"^string_of_int(numero), algoRGetoile(Noeud(y,z),s^"/*"^string_of_int
(numero),c#nothing))])
     | "epsg" ->
          let [Arete(x,Noeud(y,z))] = l in
          let numero = c#get in
          Noeud(q,[Arete(string_of_int(numero)^"|*"^s, algoRGetoile(Noeud(y,z),string_of_int(numero)
^"|*"^s,c#nothing))])
;;


(* ********************************************************************** *)
(* ********************************************************************** *)
(* ********************************************************************** *)
(* ** CREATION DU LEXIQUE ********************************************** *)
(* ********************************************************************** *)
(* ********************************************************************** *)
(* ********************************************************************** *)

(* ********************************************************************** *)
(* ** utilitaire  ****************************************************** *)
(* ** renvoie vrai si la chaine x est un chiffre ********************** *)
(* ********************************************************************** *)

let est_chiffre = function x ->
match x with
  | "0" -> true
  | "1" -> true
  | "2" -> true
  | "3" -> true
  | "4" -> true
  | "5" -> true
```

```
  | "6" -> true
  | "7" -> true
  | "8" -> true
  | "9" -> true
  | _ -> false
;;


(* ************************************************************** *)
(* ** utilitaire  ******************************************** *)
(* ** renvoie vrai si la chaine s contient un operateur / /* ... **** *)
(* ************************************************************** *)

let contient_operateur = function s ->
if contains s '/'
then true
else if contains s '|'
     then true
     else false
;;


(* ************************************************************** *)
(* ** utilitaire  ******************************************** *)
(* ** enlève les parenthèses inutiles autour d'une expression  ****** *)
(* **** exemple : (a/(b/c)) devient a/(b/c)  ********************** *)
(* ****           a/(b/c) reste inchangé     ********************** *)
(* ************************************************************** *)

let enleve_parentheses = function t ->

if (contains t '(')
then let pos = (index t '(') in
     if pos = 0
        then let l = (String.length t) in
             let ll = (l - 2) in
             (String.sub t 1 ll)
        else t
else t
;;
```

```
(* ************************************************************** *)
(* ** utilitaire  ************************************************ *)
(* ** renvoie la position de l'opérateur principal ***************** *)
(* **** exemple : ************************************************ *)
(* **** 'A' -> 0 ************************************************** *)
(* **** 'A\B' -> 1 ************************************************* *)
(* **** '(A/(B/C))\(E/F)' -> 9 ************************************** *)
(* ************************************************************** *)
(* compte_parenthese est utilisee par position_operateur_principal    *)
(* dans une chaine contenant des parenthèses :                        *)
(*   renvoie la position suivant la première parenthèse qui ferme     *)
(*   une expression bien parenthésée                                  *)
(*   exemple : compte_parenthese("A/(B/C))\(E/F)",0,[0]) = 9          *)

let rec compte_parenthese = function categorie, pos, l ->
if  l = []
then pos + 1
else if contains categorie '('
     then let pos1 = index categorie '(' in
          if contains categorie ')'
          then let pos2 = index categorie ')' in
               let longueur = String.length categorie in
               if pos1 < pos2
               then let debut = pos1+1 in
                    let long = longueur - pos1 -1 in
                    let souschaine = String.sub categorie debut long in
                    compte_parenthese(souschaine,pos+debut,[pos1] @ l)
               else let debut = pos2+1 in
                    let long = longueur - pos2 -1 in
                    let souschaine = String.sub categorie debut long in
                    compte_parenthese(souschaine,pos+debut,tl(l))
          else 0
     else if contains categorie ')'
          then let pos1 = index categorie ')' in
               let longueur = String.length categorie in
               let long = longueur-pos1-1 in
               let debut = pos1+1 in
               let souschaine = String.sub categorie debut long in
               compte_parenthese(souschaine,pos+debut,tl(l))
          else 0
```

```
;;

(* *** *)

let position_operateur_principal = function categorie ->

if contains categorie '('
then let i = index categorie '(' in
     if i = 0
     then let longueur = (String.length categorie)-1 in
          let souschaine = String.sub categorie 1 longueur in
          compte_parenthese(souschaine,0,[0])
     else if contains categorie '/'
          then let i1 = index categorie '/' in
               if contains categorie '|'
               then let i2 = index categorie '|' in
                    if i1 < i2
                    then i1
                    else i2
               else i1
          else if contains categorie '|'
               then index categorie '|'
               else 0
else if contains categorie '/'
     then let pos1 = index categorie '/' in
          if contains categorie '|'
          then let pos2 = index categorie '|' in
               if pos1 < pos2
               then pos1
               else pos2
          else pos1
     else if contains categorie '|'
          then index categorie '|'
          else 0
;;


(* ************************************************************** *)
(* ** Création du lexique ***************************************** *)
(* **** Renvoie une liste avec toutes les feuilles et leurs types *** *)
(* ************************************************************** *)
```

```
let rec creer_lexique = function Noeud(q,l), etiq, lexiq ->
if q = "[]"
then (* c'est une feuille déchargée d'arrêt *)
     lexiq
else if l = []
     then (* c'est une feuille : ajouter au lexique *)
          if contient_operateur(etiq)
          then let pos = position_operateur_principal(etiq) in
               if ( pos >= String.length etiq )
               then let nouv_etiq = enleve_parentheses(etiq) in
                    lexiq @ [ Mot(q,nouv_etiq) ]
               else lexiq @ [ Mot(q,etiq) ]
          else lexiq @ [ Mot(q,etiq) ]
     else if tl(l) = []
          then let Noeud(q,[Arete(x,Noeud(y,z))]) = Noeud(q,l) in
               creer_lexique(Noeud(y,z),x,lexiq)
          else let Noeud(q,[Arete(x,Noeud(y,z));Arete(t,Noeud(m,p))]) = Noeud(q,l) in
               let lexiq = creer_lexique(Noeud(y,z),x,lexiq) in
               creer_lexique(Noeud(m,p),t,lexiq)
;;


(* ********************************************************************** *)
(* ********************************************************************** *)
(* ********************************************************************** *)
(* ** UNIFICATION ***************************************************** *)
(* ********************************************************************** *)
(* ********************************************************************** *)
(* ********************************************************************** *)

(* ********************************************************************** *)
(* ** utilitaire ****************************************************** *)
(* ** renvoie : l'étiquette du 1er doublon ************************** *)
(* **            "" si pas de doublon      ************************** *)
(* ********************************************************************** *)

let rec renvoie_type_doublon = function l, Mot(m,e) ->
if l = []
then ""
else let Mot(q,p) = hd(l) in
```

```
     if m = q
     then p
     else renvoie_type_doublon(tl(l),Mot(m,e))
;;


(* ************************************************************** *)
(* ** utilitaire ********************************************** *)
(* ** vérifie s'il existe des doublons dans une liste ********** *)
(* ************************************************************** *)

let rec existe_doublon = function l ->
if tl(l) = []
then false
else if renvoie_type_doublon(tl(l),hd(l)) = ""
     then existe_doublon(tl(l))
     else true
;;


(* ************************************************************** *)
(* ** utilitaire ********************************************** *)
(* ** supprime les doublons d'une liste *************************** *)
(* **** Rq : un doublon = 2 mots identiques avec le même type ******* *)
(* ************************************************************** *)

let rec supprime_doublon = function l ->
if tl(l) = []
then l
else if renvoie_type_doublon(tl(l),hd(l)) = ""
         then [hd(l)]@ supprime_doublon(tl(l))
         else let Mot(p,q) = hd(l) in
              if q = renvoie_type_doublon(tl(l),hd(l))
                   then supprime_doublon(tl(l))
                   else [hd(l)]@ supprime_doublon(tl(l))
;;


(* ************************************************************** *)
(* ** utilitaire  ********************************************** *)
(* ** vérifie l'appartenance d'une sous_chaine à une chaine  ******** *)
```

```
(* **** rq : '3' est sous-chaine de '3/12' mais pas de '13/12' ****** *)
(* ************************************************************** *)

let rec est_sous_chaine = function chaine, sous_chaine, indice ->

let longueur_sous_chaine = (String.length sous_chaine) in
let indice_fin = ((String.length chaine) - longueur_sous_chaine) in
if indice <= indice_fin
then if String.sub chaine indice longueur_sous_chaine = sous_chaine
     then let indprec = (indice - 1) in
          let indsuiv = (indice + longueur_sous_chaine) in
          if (indprec >= 0)
          then let carpred = (String.sub chaine indprec 1) in
               if est_chiffre(carpred)
               then false
               else if ( indsuiv < String.length chaine )
               then let carsuiv = String.sub chaine indsuiv 1 in
                    if est_chiffre(carsuiv)
                    then false
                    else true
               else true
          else if ( indsuiv < String.length chaine )
               then let carsuiv = String.sub chaine indsuiv 1 in
                    if est_chiffre(carsuiv)
                    then false
                    else true
               else true
     else est_sous_chaine(chaine,sous_chaine,indice+1)
else false
;;


(* ************************************************************** *)
(* ** utilitaire  ******************************************** *)
(* ** renvoie l'opérateur principal d'une catégorie ************** *)
(* ************************************************************** *)

let renvoie_operateur_principal = function t ->

if contient_operateur(t)
then let pos = position_operateur_principal(t) in
```

```
     let pos2 = pos+1 in
     if ((String.sub t pos2 1) = "*")
     then String.sub t pos 2
     else String.sub t pos 1
else failwith("renvoie op princ : pas d'op !!")
;;


(* ************************************************************** *)
(* ** utilitaire  ******************************************** *)
(* ** compte le nombre d'opérateurs d'une catégorie ************** *)
(* ************************************************************** *)

let rec compte_operateurs = function t ->

if contient_operateur(t)
then if contains t '/'
          then let i1 = index t '/' in
               if contains t '|'
               then let i2 = index t '|' in
                    if i1 < i2
                    then let deb = i1 + 1 in
                         let lg = (String.length t) - i1 -1 in
                         1 + compte_operateurs(String.sub t deb lg)
                    else let deb = i2 + 1 in
                         let lg = (String.length t) - i2 -1 in
                         1 + compte_operateurs(String.sub t deb lg)
               else let deb = i1 + 1 in
                    let lg = (String.length t) - i1 -1 in
                    1 + compte_operateurs(String.sub t deb lg)
          else if contains t '|'
               then let i2 = index t '|' in
                    let deb = i2 + 1 in
                    let lg = (String.length t) - i2 -1 in
                    1 + compte_operateurs(String.sub t deb lg)
               else 0
else (* cas d'arret *)
     0
;;
```

```
(* ************************************************************** *)
(* ** utilitaire  ********************************************** *)
(* ** renvoie VRAI si la chaine ch apparait dans la liste de décomp   *)
(* ************************************************************** *)

let rec apparait_dans_liste = function ch, l_decomp ->

if l_decomp = []
then false
else if hd(l_decomp) = ch
     then true
     else apparait_dans_liste(ch,tl(l_decomp))
;;


(* ************************************************************** *)
(* ** sert pour la substitution ******************************** *)
(* **** remplace toutes les sous-chaines données par une autre chaine *)
(* ************************************************************** *)

let rec substituer_type = function chaine, sous_chaine, remplacant, indice ->

let longueur_remplacant = (String.length remplacant) in
let longueur_chaine = (String.length chaine) in
let longueur_sous_chaine = (String.length sous_chaine) in
let indice_fin = (longueur_chaine - longueur_sous_chaine) in
if (indice <= indice_fin)
then if ((String.sub chaine indice longueur_sous_chaine) = sous_chaine)
     then let nouveau_indice_debut = (indice + longueur_sous_chaine) in
          let longueur_reste = (longueur_chaine - nouveau_indice_debut) in
          let indprec = (indice - 1) in
          let indsuiv = (indice + longueur_sous_chaine) in
          if (indprec >= 0)
          then let carpred = (String.sub chaine indprec 1) in
               if est_chiffre(carpred)
               then substituer_type(chaine,sous_chaine,remplacant,indice+longueur_remplacant)
               else if (indsuiv < String.length chaine)
                    then let carsuiv = (String.sub chaine indsuiv 1) in
                         if est_chiffre(carsuiv)
                         then substituer_type(chaine,sous_chaine,remplacant,indice+longueur_remplacant)
                         else if contient_operateur(remplacant)
```

```
&&  not(contient_operateur(sous_chaine))
                              then (* on ajoute un parenthésage *)
                                   let nouvelle_chaine = (String.sub chaine 0 indice ^ "(" ^ remplacant ^ ")"
^ String.sub chaine nouveau_indice_debut longueur_reste) in
                                   substituer_type
(nouvelle_chaine,sous_chaine,remplacant,indice+longueur_remplacant)
                              else let nouvelle_chaine = (String.sub chaine 0 indice ^ remplacant ^ String.sub
chaine nouveau_indice_debut longueur_reste) in
                                   substituer_type
(nouvelle_chaine,sous_chaine,remplacant,indice+longueur_remplacant)
                    else if contient_operateur(remplacant)
&&  not(contient_operateur(sous_chaine))
                         then let nouvelle_chaine = (String.sub chaine 0 indice ^ "(" ^ remplacant ^ ")" ^
String.sub chaine nouveau_indice_debut longueur_reste) in
                              substituer_type
(nouvelle_chaine,sous_chaine,remplacant,indice+longueur_remplacant)
                         else let nouvelle_chaine = (String.sub chaine 0 indice ^ remplacant ^ String.sub
chaine nouveau_indice_debut longueur_reste) in
                              substituer_type
(nouvelle_chaine,sous_chaine,remplacant,indice+longueur_remplacant)

          else if (indsuiv < String.length chaine)
               then let carsuiv = (String.sub chaine indsuiv 1) in
                    if est_chiffre(carsuiv)
                    then substituer_type(chaine,sous_chaine,remplacant,indice+longueur_remplacant)
                    else if contient_operateur(remplacant)
&&  not(contient_operateur(sous_chaine))
                         then let nouvelle_chaine = (String.sub chaine 0 indice ^ "(" ^ remplacant ^ ")" ^
String.sub chaine nouveau_indice_debut longueur_reste) in
                              substituer_type
(nouvelle_chaine,sous_chaine,remplacant,indice+longueur_remplacant)
                         else let nouvelle_chaine = (String.sub chaine 0 indice ^ remplacant ^ String.sub
chaine nouveau_indice_debut longueur_reste) in
                              substituer_type
(nouvelle_chaine,sous_chaine,remplacant,indice+longueur_remplacant)
               else if contient_operateur(remplacant)
&&  not(contient_operateur(sous_chaine))
                    then let nouvelle_chaine = (String.sub chaine 0 indice ^ "(" ^ remplacant ^ ")" ^
String.sub chaine nouveau_indice_debut longueur_reste) in
                         substituer_type(nouvelle_chaine,sous_chaine,remplacant,indice+longueur_remplacant)
                    else let nouvelle_chaine = (String.sub chaine 0 indice ^ remplacant ^ String.sub chaine
```

```
nouveau_indice_debut longueur_reste) in
                          substituer_type(nouvelle_chaine,sous_chaine,remplacant,indice+longueur_remplacant)
     else substituer_type(chaine,sous_chaine,remplacant,indice+1)
else chaine
;;


(* ************************************************************** *)
(* ** sert pour la substitution ********************************** *)
(* **** substitution d'un type par un autre dans tout le lexique **** *)
(* ************************************************************** *)

let rec substitution = function l, type_a_remplacer, remplacant ->
if l = []
then l
else let Mot(p,q) = hd(l) in
     if q = type_a_remplacer
     then [Mot(p,remplacant)]@ substitution(tl(l),type_a_remplacer,remplacant)
     else if est_sous_chaine(q,type_a_remplacer,0)
           then if q = remplacant
                then [Mot(p,q)]@ substitution(tl(l),type_a_remplacer,remplacant)
                else [Mot(p,substituer_type(q,type_a_remplacer,remplacant,0))]@ substitution(tl
(l),type_a_remplacer,remplacant)
           else [Mot(p,q)]@ substitution(tl(l),type_a_remplacer,remplacant)
;;


(* ************************************************************** *)
(* ** sert pour la substitution ********************************** *)
(* **** Décomposition des types pour trouver les substitutions ****** *)
(* ************************************************************** *)

let decomp_en_deux = function  t ->

if contient_operateur(t)
then let pos = position_operateur_principal(t) in
     let chg = string_before t pos in
     if contains t '*'
     then let pos2 = pos+1 in
          let etoi = String.sub t pos2 1 in
          if (etoi = "*")
```

```
          then let pos3 = pos2 + 1 in
               let chd = string_after t pos3 in
               let sschg = enleve_parentheses(chg) in
               let sschd = enleve_parentheses(chd) in
               [sschg;sschd]
          else let chd = string_after t pos2 in
               let sschg = enleve_parentheses(chg) in
               let sschd = enleve_parentheses(chd) in
               [sschg;sschd]
     else let pos2 = pos+1 in
          let chd = string_after t pos2 in
          let sschg = enleve_parentheses(chg) in
          let sschd = enleve_parentheses(chd) in
          [sschg;sschd]
else []
;;


let rec decompose_type = function l ->

if l=[]
then l
else let listajout = decomp_en_deux(hd(l)) in
     [hd(l)] @ decompose_type( tl(l) @ listajout )
;;


(* ************************************************************** *)
(* ** sert pour l'unification  ********************************** *)
(* ** remplace, dans une liste de décomposition, ***************** *)
(* **** une sous chaine par une autre ***************************** *)
(* ************************************************************** *)

let rec remplacer = function l_decomp, aremp, remp ->

if l_decomp=[]
then l_decomp
else if est_sous_chaine(hd(l_decomp), aremp, 0)
     then [substituer_type(hd(l_decomp),aremp,remp,0)] @ remplacer(tl(l_decomp),aremp,remp)
     else [hd(l_decomp)] @ remplacer(tl(l_decomp),aremp,remp)
;;
```

```
(* ********************************************************************** *)
(* ** substitution d'un type par un autre ************************** *)
(* **** en matière d'unification : c'est le nerf de la guerre ! ***** *)
(* ********************************************************************** *)

let rec uni = function l, aremp, remp ->

match (aremp,remp) with
 | ( _  , [] )
 | ( [] , _  ) -> l
 | _ -> if hd(aremp) = hd(remp)              (*  x = x  => on supprime l'équation *)
        then uni( l, tl(aremp) , tl(remp) )
        else
        ( let contient_op_aremp = contient_operateur( hd(aremp) ) in
         let contient_op_remp  = contient_operateur( hd( remp) ) in
         match (contient_op_aremp,contient_op_remp) with
          | (true, true ) ->( let op_aremp = renvoie_operateur_principal(hd(aremp)) in
                              let op_remp  = renvoie_operateur_principal(hd( remp)) in
                              let egal_op  = ( op_aremp = op_remp ) in
                              match egal_op with
                               | true  -> (* même opérateur principal *)
                                          ( let aremp_app_liste = apparait_dans_liste( hd(aremp), tl(aremp)@tl
(remp) ) in
                                            match aremp_app_liste with
                                             | true  ->( let aremp_ssch_remp = est_sous_chaine( hd(remp), hd
(aremp), 0 ) in
                                                         match aremp_ssch_remp with
                                                          | true  -> failwith("echec unification : "^hd(aremp)
^" est ssch de "^hd(remp))
                                                          | false -> let nouveau_remp = [hd(remp)]@remplacer(
tl(remp), hd(aremp), hd(remp) ) in
                                                                     let nouveau_aremp = [hd(aremp)]@remplacer
( tl(aremp), hd(aremp), hd(remp) ) in
                                                                     let premier_aremp =  hd(nouveau_aremp) in
                                                                     if contains premier_aremp 'S'
                                                                     then (* celui qui contient 'S' ne doit
pas changer *)
                                                                          uni(substitution(l,hd
(nouveau_remp),hd(nouveau_aremp)),tl(nouveau_remp),tl(nouveau_aremp))
```

```
                                                                    else uni(substitution(l,hd
(nouveau_aremp),hd(nouveau_remp)),tl(nouveau_aremp),tl(nouveau_remp))
                                                                    )
                                                 | false -> let premier_aremp =  hd(aremp) in
                                                           if contains premier_aremp 'S'
                                                           then uni(substitution(l,hd(remp),hd(aremp)),tl
(remp),tl(aremp))
                                                           else uni(substitution(l,hd(aremp),hd(remp)),tl
(aremp),tl(remp))
                                               )
                                  | false ->(* opérateur principal différent : gestion des itérations *)
                                            ( let iteration = (( op_remp = "|*" ) || ( op_remp = "/*" )) in
                                              match iteration with
                                               | true  -> uni(substitution(l,hd(aremp),hd(remp)),tl(aremp),tl(tl
(tl(remp))))
                                               | false -> failwith("echec unification : op princ diff : "^hd
(aremp)^" et "^hd(remp))
                                             )
                              )

          | (false,true ) ->( let aremp_app_liste = apparait_dans_liste( hd(aremp), tl(aremp)@tl(remp) ) in
                              match aremp_app_liste with
                               | true  ->( let aremp_ssch_remp = est_sous_chaine( hd(remp), hd(aremp), 0 ) in
                                           match aremp_ssch_remp with
                                            | true  -> failwith("echec unification : "^hd(aremp)^" est ssch de
"^hd(remp))
                                            | false -> let nouveau_remp = [hd(remp)]@remplacer( tl(remp), hd
(aremp), hd(remp) ) in
                                                       let nouveau_aremp = [hd(aremp)]@remplacer( tl(aremp),
hd(aremp), hd(remp) ) in
                                                       if hd(nouveau_aremp) = "S"
                                                       then uni(substitution(l,hd(nouveau_remp),hd
(nouveau_aremp)),tl(nouveau_remp),tl(nouveau_aremp))
                                                       else uni(substitution(l,hd(nouveau_aremp),hd
(nouveau_remp)),tl(nouveau_aremp),tl(nouveau_remp))
                                          )
                               | false -> if hd(aremp) = "S"
                                          then uni(substitution(l,hd(remp),hd(aremp)),tl(remp),tl(aremp))
                                          else uni(substitution(l,hd(aremp),hd(remp)),tl(aremp),tl(remp))
                             )
          | (true, false) -> (* t = x où t n'est pas une variable : inversion de l'équation *)
```

```
                              uni( l, remp , aremp )

          | (false,false) ->(* deux variables *)
                             ( let aremp_app_liste = apparait_dans_liste( hd(aremp), tl(aremp)@tl(remp) ) in
                               match aremp_app_liste with
                               | true  ->( let aremp_ssch_remp = est_sous_chaine( hd(remp), hd(aremp), 0 ) in
                                           match aremp_ssch_remp with
                                            | true  -> failwith("echec unification : "^hd(aremp)^" est ssch de
"^hd(remp))
                                            | false -> let nouveau_remp = [hd(remp)]@remplacer( tl(remp), hd
(aremp), hd(remp) ) in
                                                       let nouveau_aremp = [hd(aremp)]@remplacer( tl(aremp), hd
(aremp), hd(remp) ) in
                                                       if hd(nouveau_aremp) = "S"
                                                       then uni(substitution(l,hd(nouveau_remp),hd
(nouveau_aremp)),tl(nouveau_remp),tl(nouveau_aremp))
                                                       else uni(substitution(l,hd(nouveau_aremp),hd
(nouveau_remp)),tl(nouveau_aremp),tl(nouveau_remp))
                                          )
                                | false -> if hd(aremp) = "S"
                                           then uni(substitution(l,hd(remp),hd(aremp)),tl(remp),tl(aremp))
                                           else uni(substitution(l,hd(aremp),hd(remp)),tl(aremp),tl(remp))
                             )
       )
;;


let unif = function l, type_a_remplacer, remplacant ->

let dec_type_a_remplacer = decompose_type([type_a_remplacer]) in
let dec_remplacant = decompose_type([remplacant]) in
uni(l,dec_type_a_remplacer,dec_remplacant)
;;


(* ********************************************************************* *)
(* ** Algorithme d'unification ***************************************** *)
(* ********************************************************************* *)

let rec unification = function l ->
if existe_doublon(l)
```

```
then let Mot(p,q) = hd(l) in
     let type_remplacant = renvoie_type_doublon(tl(l),Mot(p,q)) in
     if type_remplacant = ""
     then (* pas de substitution à effectuer *)
          unification(tl(l) @ [hd(l)])
     else (* il faut effectuer les substitutions ! *)
          unification(supprime_doublon(unif(l,q,type_remplacant)))
else (* cas d'arrêt : il n'y a pas (ou plus) de doublons *)
     l
;;


(* ********************************************************************* *)
(* ********************************************************************* *)
(* ** Application de l'apprentissage ********************************** *)
(* ***** à une liste d'arbres ******************************************* *)
(* ********************************************************************* *)
(* ********************************************************************* *)
(* ********************************************************************* *)

let rec grammaireAB = function listarb ->
if tl(listarb) = []
then unification(creer_lexique( algoRG(hd(listarb),"S",c#reset)," ",[] ))
else unification(creer_lexique( algoRG(hd(listarb),"S",c#incr)," ",[] ) @ grammaireAB(tl(listarb)))
;;

let rec grammaireABetoile = function listarb ->
if tl(listarb) = []
then unification( creer_lexique( algoRGetoile(hd(listarb),"S", c#reset), " " ,[]))
else unification( creer_lexique( algoRGetoile(hd(listarb),"S",c#incr)," " ,[]) @ grammaireABetoile(tl
(listarb)) )
;;

let rec lambek = function listarb ->
if tl(listarb) = []
then unification( creer_lexique( algoRLG(argument(normalisation(hd(listarb)),c#reset),"S"), " " ,[]))
else unification( creer_lexique( algoRLG(argument(normalisation(hd(listarb)),c#incr),"S")," " ,[]) @ lambek(tl
(listarb)) )
;;
```

```
(* ********************************************************************** *)
(* ** Affichage sur la sortie standard ********************************** *)
(* ********************************************************************** *)

let rec afficher = function lex ->
if tl(lex) = []
then let Mot(m,t) = hd(lex) in
      print_string(m); print_string(":"); print_string(t); print_string("\n");
else let Mot(m,t) = hd(lex) in
      print_string(m); print_string(":"); print_string(t); print_string("\n");
      afficher(tl(lex))
;;

(* ********************************************************************** *)
(* ** sortie fichier **************************************************** *)
(* ********************************************************************** *)

let rec sortie_fichier = function s, lex ->
if tl(lex) = []
then let Mot(m,t) = hd(lex) in
      output_string s m; output_string s ":"; output_string s t; output_string s "\n";
else let Mot(m,t) = hd(lex) in
      output_string s m; output_string s ":"; output_string s t; output_string s "\n";
      sortie_fichier(s, tl(lex))
;;

(* ********************************************************************** *)
(* ** lancer l'algo ! *************************************************** *)
(* ********************************************************************** *)

let appliquer_algo = function algo, exemple, fich_sortie ->
 let flux = open_out fich_sortie in
match algo with
| "RG"  ->     let resultat = grammaireAB(exemple) in
     afficher(resultat); sortie_fichier(flux,resultat) ; close_out flux
| "RG*" ->     let resultat = grammaireABetoile(exemple ) in
     afficher(resultat); sortie_fichier(flux,resultat) ; close_out flux
| "RLG" ->     let resultat = lambek(exemple) in
     afficher(resultat); sortie_fichier(flux,resultat) ; close_out flux

;;
```

```
(* ********************************************************************** *)
(* ********************************************************************** *)
(* ** Entrer des exemples ********************************************** *)
(* ********************************************************************** *)
(* ********************************************************************** *)

let rec entrerexemple = function algo, n ->
  let top = openTk () in  Wm.title_set top "Exemple : ";
  let entree = Entry.create top in pack[entree];
  let entrer () = entrerexemple(algo, n) in output n "       " 0 5;
  let close () = let a = String.length (Entry.get entree) in
  output n (Entry.get entree) 0 a in
  let button1 = Button.create ~text: "Enregistrer" ~command:close top in
  let button2 = Button.create ~text: "Autre exemple" ~command:entrer top in
  let button3 = Button.create ~text: "Appliquer l'algorithme " (*~command:*) top in
  pack [button1;button2;button3];
  mainLoop()

;;

let exemple = function algo, n ->
  let oc = open_out n in
  entrerexemple(algo, oc);
  close_out oc ;
  mainLoop()

;;
(* ********************************************************************** *)
(* ********************************************************************** *)
(* ** Nom du fichier a charger ***************************************** *)
(* ********************************************************************** *)
(* ********************************************************************** *)

let nom = function algo ->
  let main_window = openTk () in Wm.title_set main_window "nom du fichier";
  let entree = Entry.create main_window in  pack[entree];
  let name () = exemple(algo,(Entry.get entree)) in
  let quit = Button.create  ~text: "Charger" ~command:name  main_window in  pack[quit];
  mainLoop()
;;
```

```
(* ********************************************************************** *)
(* ********************************************************************** *)
(* ** Charger un fichier XML deja existant ************************** *)
(* ********************************************************************** *)
(* ********************************************************************** *)

(* ////////////////////// Affichage des arbres \\\\\\\\\\\\\\\\\\\\\\ *)

let env = new environnement_graphique;;

let aff = function fich_entree ->
  let doc = parse_document_entity default_config (from_file fich_entree) spec in
  let arbre = creer_arbre doc in  env#debut_trace;
  tracer_arbre env 20 20 arbre;
  env#fin_trace
;;


let fin () = close_graph()
;;

(* \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\ *)
(* ////////////////////// Charger un document \\\\\\\\\\\\\\\\\\\\\\ *)

let doc2listarbre = function algo, fich_entree, fich_sortie ->
  let doc = parse_document_entity default_config (from_file fich_entree) spec in
  let arbre = creer_arbre doc in
  appliquer_algo(algo,arbre,fich_sortie)
;;

(* \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\ *)

let charger = function algo ->
  let top = openTk () in Wm.title_set top "Nom du fichier XML";
  let entre1 = Entry.create top in  let entre2 = Entry.create top in  pack [entre1; entre2];
  let charg     () = doc2listarbre(algo,Entry.get entre1, Entry.get entre2) in
  let affichage () = aff(Entry.get entre1) in
  let fermeture () = fin() in
  let button1  = Button.create ~text:"              Appliquer l'algorithme               " ~command:charg
top in
```

```
  let button2  = Button.create ~text:"                  Afficher arbre                  "
~command:affichage top in
  let button3  = Button.create ~text:"               Fermer affichage arbre             "
~command:fermeture top in
  pack[button1;button2;button3];
  mainLoop()
;;

(* ********************************************************************* *)
(* ********************************************************************* *)
(* ** Choix pour l'entree des donnees ********************************** *)
(* ********************************************************************* *)
(* ********************************************************************* *)

let entree = function algo ->
  let titre = "Algorithme "^algo in
  let top = openTk () in Wm.title_set top titre;
  let charge () = charger(algo) in
  let name () = nom(algo) in
  let button1 = Button.create ~text: "          Charger un fichier deja existant            " ~command:charge
top in
  let button2 =   Button.create ~text: " Entrer des exemples dans un fichier (pas implémenté) " ~command:name
top in
  pack [button1;button2];
  mainLoop()
;;

(* ********************************************************************* *)
(* ********************************************************************* *)
(* ** Menu d'entrée **************************************************** *)
(* ********************************************************************* *)
(* ********************************************************************* *)

let programme () =
  let main_window = openTk () in Wm.title_set main_window "Choix de l'algorithme d'apprentissage";
  let entree1 () = entree("RG") in
  let entree2 () = entree("RG*") in
  let entree3 () = entree("RLG") in
  let button1 =   Button.create ~text: "                    Algorithme RG                   "
~command:entree1 main_window in
  let button2 =   Button.create ~text: "                    Algorithme RG*                  "
```

```
~command:entree2 main_window in
  let button3 =   Button.create ~text: "                    Algorithme RLG                    "
~command:entree3 main_window in
  pack [button1;button2;button3];
  mainLoop ()

;;
```